

GI-Edition



**Lecture Notes
in Informatics**

**Theo Härder, Wolfgang Lehner,
Bernhard Mitschang, Harald Schöning,
Holger Schwarz (Hrsg.)**

Datenbanksysteme für Business, Technologie und Web (BTW)

**14. Fachtagung des GI-Fachbereichs
„Datenbanken und Informationssysteme“
(DBIS)**

**02. - 04.03.2011
in Kaiserslautern, Germany**

Proceedings



Theo Härder, Wolfgang Lehner, Bernhard Mitschang,
Harald Schöning, Holger Schwarz (Hrsg.)

**Datenbanksysteme für
Business, Technologie und Web
(BTW)**

**14. Fachtagung des GI-Fachbereichs
„Datenbanken und Informationssysteme“ (DBIS)**

**02. – 04.03.2011
in Kaiserslautern, Germany**

Gesellschaft für Informatik e.V. (GI)

Lecture Notes in Informatics (LNI) - Proceedings

Series of the Gesellschaft für Informatik (GI)

Volume P-180

ISBN 978-3-88579-274-1

ISSN 1617-5468

Volume Editors

Theo Härder

Fachbereich Informatik
Universität Kaiserslautern
67653 Kaiserslautern, Germany
E-Mail: haerder@informatik.uni-kl.de

Wolfgang Lehner

Department of Computer Science
Technische Universität Dresden
01187 Dresden, Germany
E-Mail: wolfgang.lehner@tu-dresden.de

Bernhard Mitschang

Institut für Parallele und Verteilte Systeme (IPVS)
Universität Stuttgart
70569 Stuttgart, Germany
E-Mail: bernhard.mitschang@ipvs.uni-stuttgart.de

Harald Schöning

Software AG
64297 Darmstadt, Germany
E-Mail: harald.schoening@softwareag.com

Holger Schwarz

Institut für Parallele und Verteilte Systeme (IPVS)
Universität Stuttgart
70569 Stuttgart, Germany
E-Mail: holger.schwarz@ipvs.uni-stuttgart.de

Series Editorial Board

Heinrich C. Mayr, Universität Klagenfurt, Austria (Chairman, mayr@ifit.uni-klu.ac.at)

Hinrich Bonin, Leuphana-Universität Lüneburg, Germany

Dieter Fellner, Technische Universität Darmstadt, Germany

Ulrich Flegel, SAP Research, Germany

Ulrich Frank, Universität Duisburg-Essen, Germany

Johann-Christoph Freytag, Humboldt-Universität Berlin, Germany

Thomas Roth-Berghofer, DFKI

Michael Goedicke, Universität Duisburg-Essen

Ralf Hofestädt, Universität Bielefeld

Michael Koch, Universität der Bundeswehr, München, Germany

Axel Lehmann, Universität der Bundeswehr München, Germany

Ernst W. Mayr, Technische Universität München, Germany

Sigrid Schubert, Universität Siegen, Germany
Martin Warnke, Leuphana-Universität Lüneburg, Germany

Dissertations

Steffen Hölldobler, Technische Universität Dresden, Germany

Seminars

Reinhard Wilhelm, Universität des Saarlandes, Germany

Thematics

Andreas Oberweis, Universität Karlsruhe (TH)

© Gesellschaft für Informatik, Bonn 2011

printed by Köllen Druck+Verlag GmbH, Bonn

Vorwort

Die 14. BTW-Tagung der Gesellschaft für Informatik (GI) fand vom 2. bis 4. März 2011 an der Technischen Universität Kaiserslautern statt. In den letzten beiden Jahrzehnten hat sich Kaiserslautern dank seiner Hochschulen und Forschungseinrichtungen zu einem leistungsstarken Zentrum für innovative Technologieprodukte und Dienstleistungen entwickelt. Wesentlich dazu beigetragen hat die Technische Universität Kaiserslautern, die 2010 ihren 40. Geburtstag feierte. Die stürmische Entwicklung im IT-Bereich wurde, neben dem Fachbereich Informatik, vor allem durch die Fraunhofer-Institute für Experimentelles Software Engineering (IESE), das Institut für Techno- und Wirtschaftsmathematik (ITWM) sowie das Deutsche Forschungszentrum für Künstliche Intelligenz (DFKI) vorangetrieben. Durch die jüngste Einrichtung eines Max-Planck-Instituts für Software-Systeme erfährt die Wissenschaftsstadt Kaiserslautern eine weitere, starke Aufwertung im internationalen Forschungsumfeld.

Die BTW-Tagung ist nun seit über 25 Jahren das zentrale Forum der deutschsprachigen Datenbankgemeinde. Auf dieser Tagung treffen sich alle zwei Jahre nicht nur Wissenschaftler, sondern auch Praktiker und Anwender, die sich zu den vielfältigen Themen der Datenbank- und Informationssystemtechnologie austauschen. So wie sich die Eigenschaften der zu organisierenden und zu verwaltenden Daten verändert haben, haben sich auch die Techniken zu deren Organisation und Verarbeitung verändert und den neuen Herausforderungen angepasst. Neben der Behandlung strukturierter Daten nehmen in den letzten Jahren semi- bzw. unstrukturierte Daten einen immer größeren Raum ein. Klassische, zentrale Datenbanksystem-Architekturen wurden erweitert und teilweise abgelöst von hauptspeicherbasierten, verteilten, parallelen oder offenen Systemen. Dabei spielen neue Hardware-Möglichkeiten eine immer wichtigere Rolle: hierzu gehören mobile Systeme, Multi-Core-Systeme, Graphikkarten oder auch spezielle Speichermedien wie Flash-Speicher. Diese Techniken spiegeln sich in den aktuellen Themenbrennpunkten wider: Informationsintegration, Datenanalyse, Ontologien und Semantic Web, Datenstromverarbeitung, Service-Orientierung, Cloud-Technologien, Virtualisierungstechniken, Energieeffizienz u.v.a.m.

In guter Tradition umfasste auch die BTW 2011 ein wissenschaftliches Programm, ein Industrieprogramm und ein Demonstrationsprogramm sowie ein Studierendenprogramm, verschiedene Workshops und auch Tutorien zu aktuellen Themen im Rahmen der Datenbank-Tutorientage. Frühere BTW-Tagungen erreichten stets etwa 80 Einreichungen, aus denen nach strikter und anonymer Begutachtung das Tagungsprogramm ausgewählt wurde. Durch die Spezialisierung vieler traditioneller BTW-Themenbereiche, die heutzutage mit eigenen Tagungen und Workshops um Beiträge werben, reduzierte sich die Anzahl der Einreichungen bei den zuletzt durchgeführten BTW-Tagungen merklich. Umso

beachtlicher ist der Wettbewerb um die Teilnahme an der BTW 2011 zu bewerten, da die Anzahl der Einreichungen mit 83 (70 zum wissenschaftlichen Programm und 13 zum Industrieprogramm) für die BTW eine Rekordmarke erreichte. Zwei kompetente Programmkomitees haben daraus für das wissenschaftliche Programm 24 Lang- und 6 Kurzbeiträge und für das Industrieprogramm 6 Beiträge ausgewählt, die in diesen Tagungsband aufgenommen und während der Tagung vorgetragen wurden. Im Demonstrationsprogramm konnten von 15 Einreichungen 12 zur Präsentation angenommen werden. Alle akzeptierten Demonstrationen sind in Kurzform in diesem Tagungsband beschrieben.

Zum sechsten Mal war im Rahmen der BTW ein Wettbewerb um die beste Dissertation, diesmal für den Zeitraum Oktober 2008 bis September 2010, im Bereich der Datenbank- und Informationssysteme ausgeschrieben. Die Auszeichnung erhielt Dr. Christian Mathis für seine von Prof. Theo Härder betreute Dissertation "Storing, Indexing, and Querying XML Documents in Native Database Management Systems".

Zusätzlich umfasste das Programm der BTW 2011 mehrere eingeladene Vorträge. Das wissenschaftliche Programm wurde durch Themen aus den Bereichen Kerndatenbanktechnologie, Middleware und Entwicklung von Web-Applikationen bereichert. Vortragende waren Hasso Plattner (SAP AG und HPI Potsdam), Nelson Mattos (Google EMEA) sowie Shivakumar Vaithyanathan (IBM Research). Die Attraktivität des Industrieprogramms wurde durch drei Beiträge von Namik Hrle (IBM), Dieter Gawlik (Oracle) und Franz Färber (SAP AG) zu den Themen Datenanalyse, Ereignisverarbeitung und Multi-Mandantenfähigkeit erhöht.

Die Materialien zur BTW 2011 werden auch über die Tagung hinaus unter <http://btw2011.de> zu Verfügung stehen.

Die Organisation einer so großen Tagung wie der BTW mit ihren angeschlossenen Veranstaltungen ist nicht ohne zahlreiche Partner und Unterstützer möglich. Sie sind auf den folgenden Seiten aufgeführt. Ihnen gilt unser besonderer Dank ebenso wie den Sponsoren der Tagung und der GI-Geschäftsstelle.

Kaiserslautern, Dresden, Stuttgart, Darmstadt, im Januar 2011

Bernhard Mitschang, Vorsitzender des Programmkomitees

Harald Schöning, Vorsitzender des Industriekomitees

Wolfgang Lehner, Vorsitzender des Demonstrationskomitees

Theo Härder, Tagungsleitung und Vorsitzender des Organisationskomitees

Holger Schwarz, Tagungsband und Konferenz-Management-System

Tagungsleitung

Theo Härder, TU Kaiserslautern

Organisationskomitee

Theo Härder

Stefan Deßloch

Andreas Reuter

Sebastian Bächle

Volker Hudlet

Heike Neu

Steffen Reithermann

Daniel Schall

Karsten Schmidt

Andreas Weiner

Studierendenprogramm

Hagen Höpfner, Bauhaus-Universität Weimar

Joachim Klein, TU Kaiserslautern

Koordination Workshops

Bernhard Mitschang, Univ. Stuttgart

Stefan Deßloch, TU Kaiserslautern

Thomas Jörg, TU Kaiserslautern

Tutorientage

Gottfried Vossen, Univ. Münster

Alexander Rabe, DIA Bonn

Programmkomitees

Wissenschaftliches Programm

Vorsitz: Bernhard Mitschang, Univ. Stuttgart

Wolf-Tilo Balke, TU Braunschweig

Michael Böhlen, Univ. Zürich

Alex Buchmann, TU Darmstadt

Erik Buchmann, Karlsruher Institut
für Technologie

Stefan Dessloch, TU Kaiserslautern

Jens Dittrich, Univ. des Saarlandes

Johann-Christoph Freytag, HU Berlin

Norbert Fuhr, Univ. Duisburg Essen

Torsten Grust, Univ. Tübingen

Gregor Hackenbroich, SAP

Theo Härder, TU Kaiserslautern

Andreas Henrich, Univ. Bamberg

Carl-Christian Kanne, Univ. Mann-
heim

Daniel Keim, Univ. Konstanz

Alfons Kemper, TU München

Wolfgang Klas, Univ. Wien

Meike Klettke, Univ. Rostock

Birgitta König-Ries, Univ. Jena

Donald Kossmann, ETH Zürich

Hans-Peter Kriegel, Univ. München

Klaus Küspert, Univ. Jena

Wolfgang Lehner, TU Dresden

Ulf Leser, HU Berlin
Frank Leymann, Univ. Stuttgart
Volker Linnemann, Univ. Lübeck
Thomas Mandl, Univ. Hildesheim
Rainer Manthey, Univ. Bonn
Volker Markl, TU Berlin
Klaus Meyer-Wegener, Univ. Erlangen-Nürnberg
Felix Naumann, HPI Potsdam
Daniela Nicklas, Univ. Oldenburg
Peter Peinl, Hochschule Fulda
Erhard Rahm, Univ. Leipzig
Manfred Reichert, Univ. Ulm
Norbert Ritter, Univ. Hamburg

Gunter Saake, Univ. Magdeburg
Kai-Uwe Sattler, TU Ilmenau
Eike Schallehn, Univ. Magdeburg
Ingo Schmitt, BTU Cottbus
Hinrich Schütze, Univ. Stuttgart
Holger Schwarz, Univ. Stuttgart
Bernhard Seeger, Univ. Marburg
Thomas Seidl, RWTH Aachen
Knut Stolze, IBM
Uta Störl, Hochschule Darmstadt
Gottfried Vossen, Univ. Münster
Gerhard Weikum, MPI Saarbrücken
Mathias Weske, Univ. Potsdam

Industrieprogramm

Vorsitz: Harald Schöning, Software AG

Goetz Graefe, HP
Klaudia Hergula, Daimler
Albert Maier, IBM

Christian Mathis, SAP
Berthold Reinwald, IBM
Thomas Ruf, GfK

Demonstrationsprogramm

Wolfgang Lehner, TU Dresden

Rainer Gemulla, MPI Saarbrücken

Gutachter für Dissertationspreise

Johann-Christoph Freytag, HU Berlin
Felix Naumann, HPI Potsdam

Gottfried Vossen, Univ. Münster

Externe Gutachter

Daniar Achakeyev
Stefan Appel
Christian Beecks
Thomas Bernecker
Tobias Binz
Brigitte Boden
Matthias Boehm
Thomas Boettcher
André Bolles
Alexander Borusan
Falk Brauer

Katharina Büchse
Dietrich Christopeit
Markus Döhring
Tobias Emrich
Stefan Endrullis
Stephan Ewen
Christoph Fehling
Tobias Freudenreich
Sebastian Frischbier
Sergej Fries
Ingolf Geist

Andreas Göbel
Marco Grawunder
Michael Hartung
Till Haselmann
Arvid Heise
Geerd-Dietger Hoffmann
Fabian Hüske
Anca Ivanescu
Slava Kisilevich
Hanna Köpcke
Lars Kolb
Jens Lechtenbörger
Max Lehn
Matthias Liebisch
Christoph Lofi
Andreas Lübcke
Polina Malets
Florian Mansmann
Fabian Panse
Syed Saif ur Rahman
Michael Reiter

Astrid Rheinländer
Christian Rohrdantz
Philipp Rösch
Marko Rosenmüller
Sascha Saretz
Jörg Schad
Matthias Schäfer
Benjamin Schlegel
Stefan Schuh
Sandro Schulze
Joachim Selke
Norbert Siegmund
Svenja Simon
Steve Strauch
Gunnar Thies
Florian Verhein
Michael von Riegen
Stephan Vornholt
David Zellhöfer
Arthur Zimek
Andreas Züfle

Inhaltsverzeichnis

Eingeladene Vorträge

Hasso Plattner (Hasso-Plattner-Institut Potsdam): <i>SanssouciDB: An In-Memory Database for Processing Enterprise Workloads</i>	2
Nelson Mattos (Google): <i>The Web as the development platform of the future</i>	22
Shivakumar Vaithyanathan (IBM Research): <i>The Power of Declarative Languages: From Information Extraction to Machine Learning</i>	23

Wissenschaftliches Programm

Verarbeitung großer Datenmengen

Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hüske, Odej Kao, Volker Markl, Erik Nijkamp, Daniel Warneke (Technische Universität Berlin): <i>MapReduce and PACT - Comparing Data Parallel Programming Models</i>	25
Lars Kolb, Andreas Thor, Erhard Rahm (Universität Leipzig): <i>Parallel Sorted Neighborhood Blocking with MapReduce</i>	45
Alexander Schätzle, Martin Przyjacił-Zablocki, Thomas Hornung, Georg Lausen (Universität Freiburg): <i>PigSPARQL: Übersetzung von SPARQL nach Pig Latin</i>	65

Datenströme

Dennis Geesen, André Bolles, Marco Grawunder, Jonas Jacobi, Daniela Nicklas, H.-Jürgen Appelrath (Universität Oldenburg): <i>Koordinierte zyklische Kontext-Aktualisierungen in Datenströmen</i>	85
Parisa Haghani (EPFL), Sebastian Michel (Universität des Saarlandes), Karl Aberer (EPFL): <i>Tracking Hot-k Items over Web 2.0 Streams</i>	105

Claas Busemann, Christian Kuka (OFFIS Oldenburg) Daniela Nicklas, Susanne Boll (Universität Oldenburg): <i>Flexible and Efficient Sensor Data Processing - A Hybrid Approach</i>	123
Marc Wichterich, Anca Maria Ivanescu, Thomas Seidl (RWTH Aachen): <i>Feature-Based Graph Similarity with Co-Occurrence Histograms and the Earth Mover's Distance</i>	135
 <i>Vorhersagemodelle</i>	
Sebastian Bächle, Karsten Schmidt (Technische Universität Kaiserslautern): <i>Lightweight Performance Forecasts for Buffer Algorithms</i>	147
Ulrike Fischer, Matthias Boehm, Wolfgang Lehner (Technische Universität Dresden): <i>Offline Design Tuning for Hierarchies of Forecast Models</i>	167
Maik Häsner, Conny Junghans, Christian Sengstock, Michael Gertz (Universität Heidelberg): <i>Online Hot Spot Prediction in Road Networks</i>	187
 <i>DB-Implementierung</i>	
Andreas M. Weiner (Technische Universität Kaiserslautern): <i>Advanced Cardinality Estimation in the XML Query Graph Model</i>	207
Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, Wolfgang Lehner (Technische Universität Dresden): <i>Efficient In-Memory Indexing with Generalized Prefix Trees</i>	227
Joachim Klein (Technische Universität Kaiserslautern): <i>Stets Wertvollständig! - Snapshot Isolation für das Constraint- basierte Datenbank-Caching</i>	247
 <i>Anfrageverarbeitung</i>	
Goetz Graefe (HP): <i>A generalized join algorithm</i>	267
Thomas Jörg, Stefan Dessloch (Technische Universität Kaiserslautern): <i>View Maintenance using Partial Deltas</i>	287

Francis Gropengießer, Stephan Baumann, Kai-Uwe Sattler (Technische Universität Ilmenau): <i>Cloudy Transactions: Cooperative XML Authoring on Amazon S3</i>	307
 <i>Informationsextraktion</i>	
Joachim Selke, Silviu Homocanu, Wolf-Tilo Balke (Technische Universität Braunschweig): <i>Conceptual Views for Entity-Centric Search: Turning Data into Meaningful Concepts</i>	327
Emmanuel Müller (Karlsruher Institut für Technologie), Ira Assent (Aarhus University), Stephan Günnemann, Patrick Gerwert, Matthias Hannen, Timm Jansen, Thomas Seidl (RWTH Aachen): <i>A Framework for Evaluation and Exploration of Clustering Algorithms in Subspaces of High Dimensional Databases</i>	347
Adriana Budura (EPFL), Sebastian Michel (Universität des Saarlandes), Karl Aberer (EPFL): <i>Efficient Interest Group Discovery in Social Networks using an Integrated Structure/Quality Index</i>	367
Christian Böhm (LMU München), Ines Färber, Sergej Fries (RWTH Aachen), Ulrike Korte (BSI Bonn), Johannes Merkle (secunet Essen), Annahita Oswald (LMU München), Thomas Seidl (RWTH Aachen), Bianca Wackersreuther, Peter Wackersreuther (LMU München): <i>Filtertechniken für geschützte biometrische Datenbanken</i>	379
 <i>Benchmarking & Simulation</i>	
Florian Funke, Alfons Kemper, Thomas Neumann (Technische Universität München): <i>Benchmarking Hybrid OLTP&OLAP Database Systems</i>	390
Jan Schaffner, Benjamin Eckart, Christian Schwarz, Jan Brunnert (Hasso-Plattner-Institut Potsdam), Dean Jacobs (SAP AG), Alexander Zeier, Hasso Plattner (Hasso-Plattner-Institut Potsdam): <i>Simulating Multi-Tenant OLAP Database Clusters</i>	410
Volker Hudlet, Daniel Schall (Technische Universität Kaiserslautern): <i>SSD \neq SSD - An Empirical Study to Identify Common Properties and Type-specific Behavior</i>	430

Lukas Rupperecht, Jessica Smejkal, Angelika Reiser, Alfons Kemper (Technische Universität München): <i>HiSim: A Highly Extensible Large-Scale P2P Network Simulator</i>	442
 <i>Probabilistische und inkonsistente Daten</i>	
Jochen Adamek (Technische Universität Berlin), Katrin Eisenreich (SAP AG), Volker Markl (Technische Universität Berlin), Philipp Rösch (SAP AG): <i>Operators for Analyzing and Modifying Probabilistic Data - A Question of Efficiency</i>	454
Maximilian Dylla, Mauro Sozio, Martin Theobald (Max-Planck-Institut für Informatik Saarbrücken): <i>Resolving Temporal Conflicts in Inconsistent RDF Knowledge Bases</i>	474
Sebastian Lehrack, Sascha Saretz, Ingo Schmitt (Technische Universität Cottbus): <i>QSQL^P: Eine Erweiterung der probabilistischen Many-World- Semantik um Relevanzwahrscheinlichkeiten</i>	494
 <i>Maßgeschneiderte DB-Anwendungen</i>	
Martin Schäler (Universität Magdeburg), Thomas Leich (METOP Forschungsinstitut Magdeburg), Norbert Siegmund (Universität Magdeburg), Christian Kästner (Universität Marburg), Gunter Saake (Universität Magdeburg): <i>Generierung maßgeschneiderter Relationenschemata in Software- produktlinien mittels Superimposition</i>	514
Peter Reimann, Michael Reiter, Holger Schwarz, Dimka Karastoyanova, Frank Leymann (Universität Stuttgart): <i>SIMPL – A Framework for Accessing External Data in Simulation Workflows</i>	534
Sven Efftinge (itemis AG), Sören Frey, Wilhelm Hasselbring (Universität Kiel), Jan Köhnlein (itemis AG): <i>Einsatz domänenspezifischer Sprachen zur Migration von Daten- bankanwendungen</i>	554

Dissertationspreis

Christian Mathis (SAP AG):

XML Query Processing in XTC

575

Industrieprogramm

Complex Event Processing und Reporting

Eingeladener Vortrag:

596

Diogo Guerra (CISUC/University of Coimbra),

Ute Gawlick (University of Utah Health Sciences Center),

Pedro Bizarro (CISUC/University of Coimbra),

Dieter Gawlick (Oracle Corporation):

An Integrated Data Management Approach to Manage Health Care Data

Jens Schimmelpfennig, Dirk Mayer, Philipp Walter; Christian Seel
(IDS Scheer AG):

606

Involving Business Users in the Design of Complex Event Processing Systems

Ruben Pulido de los Reyes, Christoph Sieb (IBM):

616

Fast and Easy Delivery of Data Mining Insights to Reporting Systems

Rund um OLAP

Eingeladener Vortrag:

626

Namik Hrle, Oliver Draese (IBM):

Technical Introduction to the IBM Smart Analytics Optimizer for DB2 for System z

Knut Stolze (IBM), Felix Beier (IBM, Technische Universität
Ilmenau), Kai-Uwe Sattler (Technische Universität Ilmenau),
Sebastian Sprenger, Carlos Caballero Grolimund, Marco Czech
(IBM):

628

*Architecture of a Highly Scalable Data Warehouse Appliance
Integrated to Mainframe Database Systems*

Martin Oberhofer, Michael Wurst (IBM): <i>Interactive Predictive Analytics with Columnar Databases</i>	640
 <i>In-Memory und Cloud</i>	
Eingeladener Vortrag: Franz Färber, Christian Mathis, Daniel Duane Culp, Wolfram Kleis (SAP AG): <i>An In-Memory Database System for Multi-Tenant Applications</i>	650
Christian Tinnefeld, Stephan Müller, Helen Kaltegärtner, Sebastian Hillig, Lars Butzmann, David Eickhoff, Stefan Klauck, Daniel Taschik, Björn Wagner, Oliver Xylander, Alexander Zeier, Hasso Plattner (Hasso-Plattner-Institut Potsdam), Cafer Tosun (SAP AG): <i>Available-To-Promise on an In-Memory Column Store</i>	667
Michael C. Jaeger, Uwe Hohenstein (Siemens AG): <i>Cloud Storage: Wie viel Cloud Computing steckt dahinter?</i>	687
 <i>Panel</i>	
Jens Dittrich (Universität des Saarlandes), Franz Färber (SAP AG), Goetz Graefe (HP), Henrik Loeser (IBM), Wilfried Reimann (Daimler AG), Harald Schöning (Software AG): <i>“One Size Fits All”: An Idea Whose Time Has Come and Gone?</i>	703
 Demonstrationsprogramm	
Mohammed AbuJarour, Felix Naumann (Hasso-Plattner-Institut Potsdam, Universität Potsdam): <i>Improving Service Discovery through Enriched Service Descriptions</i>	706
André Bolles, Dennis Geesen, Marcro Grawunder, Jonas Jacobi, Daniela Nicklas, Hans-Jürgen Appelrath (Universität Oldenburg), Marco Hannibal, Frank Köster (DLR Braunschweig): <i>StreamCars - Datenstrommanagementbasierte Verarbeitung von Sensordaten im Fahrzeug</i>	710
Nazario Cipriani, Carlos Lübbe, Oliver Dörler (Universität Stuttgart): <i>NexusDSEditor - Integrated Tool Support for the Data Stream Processing Middleware NexusDS</i>	714
Gereon Schüller, Andreas Behrend (Fraunhofer FKIE): <i>AIMS: An SQL-based System for Airspace Monitoring</i>	718

Christian Wartner, Sven Kitschke (Universität Leipzig): <i>PROOF: Produktmonitoring im Web</i>	722
Matthias Fischer, Marco Link, Nicole Zeise, Erich Ortner (Technische Universität Darmstadt): <i>ProCEM Software Suite - Integrierte Ablaufsteuerung und - überwachung auf Basis von Open Source Systemen</i>	726
Tilman Rabl, Hatem Mousselly Sergieh, Michael Frank, Harald Kosch (Universität Passau): <i>Demonstration des Parallel Data Generation Framework</i>	730
Volker Hudlet, Daniel Schall (Technische Universität Kaiserslautern): <i>Measuring Energy Consumption of a Database Cluster</i>	734
Jens Teubner, Louis Woods (ETH Zürich): <i>Snowfall: Hardware Stream Analysis Made Easy</i>	738
Horst Werner, Christof Bornhoevd, Robert Kubis, Hannes Voigt (SAP AG): <i>MOAW: An Agile Visual Modeling and Exploration Tool for Irregu- larly Structured Data</i>	742
Martin Hahmann, Dirk Habich, Wolfgang Lehner (Technische Universität Dresden): <i>Touch it, Mine it, View it, Shape it</i>	746
Martin Oberhofer, Albert Maier, Thomas Schwarz, Manfred Vodegel (IBM): <i>Metadata-driven Data Migration for SAP Projects</i>	750

Eingeladene Vorträge

SanssouciDB: An In-Memory Database for Processing Enterprise Workloads

Hasso Plattner
Hasso-Plattner-Institute
University of Potsdam
August-Bebel-Str. 88
14482 Potsdam, Germany
Email: hasso.plattner@hpi.uni-potsdam.de

Abstract: In this paper, we present SanssouciDB: a database system designed for serving ERP transactions and analytics out of the same data store. It consists of a column-store engine for high-speed analytics and transactions on sparse tables, as well as an engine for so-called combined columns, i.e., column groups which are used for materializing result sets, intermediates, and for processing transactions on tables touching many attributes at the same time. Our analysis of SAP customer data showed that the vast majority of transactions in an ERP system are of analytical nature. We describe the key concepts of SanssouciDB’s architecture: concurrency control, techniques for compression and parallelization, and logging. To illustrate the potential of combining OLTP and OLAP processing in the same database, we give several examples of new applications which have been built on top of an early version of SanssouciDB and discuss the speedup achieved when running these applications at SAP customer sites.

1 Introduction

The motto for the last 25 years of commercial DBMS development could well have been “One Size Fits All” [SMA⁺07]. Traditional DBMS architectures have been designed to support a wide variety of applications with general-purpose data management functionality. All of these applications have different characteristics and place different demands on the data management software. The general-purpose database management systems that rule the market today do everything well but do not excel in any area.

Directly incorporating the characteristics of certain application areas and addressing them in the system architecture as well as in the data layout can improve performance by at least a factor of ten. Such major gains were reported from database systems tailored to application areas such as text search and text mining, stream processing, and data warehousing [SMA⁺07]. In the following, we will use the term characteristic-oriented database system to refer to such systems. Our vision is to unite operational processing and analytical processing in one database management system for enterprise applications. We believe that this effort is an essential prerequisite for addressing the shortcomings of existing solutions to enterprise data management and for meeting the requirements of tomorrow’s enterprise applications (see also [Pla09]).

This paper introduces SanssouciDB, an in-memory database for processing enterprise workloads consisting of both transactional and analytical queries. SanssouciDB picks up the idea of a characteristics-oriented database system: it is specifically tailored to enterprise applications. Although the main constituents of SanssouciDB’s architecture are well-known techniques which were previously available, we combine them in a novel way.

The paper is organized as follows: Section 2 gives an overview of SanssouciDB’s architecture. Afterwards, we describe three important components of the architecture in greater detail: Section 3 provides the reader with details of how data access is organized in main memory and how compression weighs in. Section 4 describes transaction management in SanssouciDB by explaining how concurrency control is realized as well as presenting the techniques used for logging and recovery. In Section 5 we present the parallel aggregation and join algorithms that we implemented for SanssouciDB. We believe that SanssouciDB’s architecture has a great potential for improving the performance of enterprise applications. Therefore, in Section 6, we give a couple of application examples where significant improvements could be achieved at a number of SAP customer sites using the concepts presented in this paper. Section 7 concludes the paper.

2 Architecture of SanssouciDB

Nearly all enterprise applications rely on the relational data model, so we have made SanssouciDB a relational database system. The relations stored in SanssouciDB permanently reside in main memory, since accessing main memory is orders of magnitude faster than accessing disk. Figure 1 presents a conceptual overview of SanssouciDB. SanssouciDB runs on a cluster of blades in a distributed fashion, with one server process per blade. The server process itself can run multiple threads, one per physical core available on the blade, managed by a scheduler (not shown in Figure 1).

To communicate with clients and other server processes, a SanssouciDB server process has an interface service and a session manager. The session manager keeps track of client connections and the associated parameters such as the connection timeout. The interface service provides the SQL interface and support for stored procedures. The interface service runs on top of the distribution layer, which is responsible for coordinating distributed metadata handling, distributed transaction processing, and distributed query processing. To allow fast, blade-local metadata lookups, the distribution layer replicates and synchronizes metadata across the server processes running on the different blades. The metadata contains information about the storage location of tables and their partitions. Because data may be partitioned across blades, SanssouciDB provides distributed transactions and distributed query processing. The distribution layer also includes the transaction manager. While there are many interesting challenges in the distribution layer, we omit a detailed discussion of these topics in this paper. Data replication for column-oriented databases is discussed in [SEJ⁺11].

The main copy of a database table is kept in main memory (rather than on disk) and

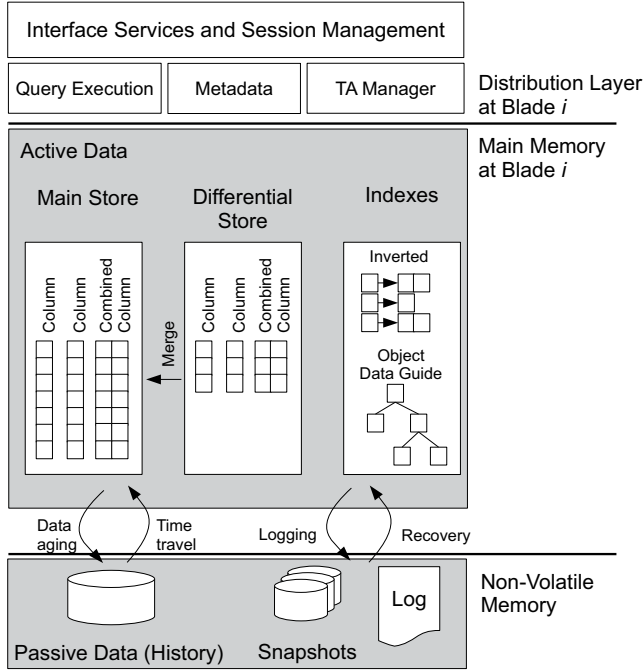


Figure 1: Conceptual Overview of SanssouciDB

consists of a main store, a differential store, and a collection of indexes. Non-volatile storage is required to provide the persistence for the database. Section 3.1 presents a detailed discussion about the separation into main and differential store.

Ideally, we would like to fit the complete database of an enterprise onto a single blade, that is, into a machine with a single main board containing multiple CPUs and a large array of main memory modules. However, not even the largest blades available at the time of writing allow us to do so. We thus assume a cluster of multiple blades, where the blades are interconnected by a network (see Figure 1).

A necessary prerequisite for a database system running on such a cluster of blades is data partitioning and distribution across blades. Managing data across blades introduces more complexity into the system, for example, distributed query processing algorithms accessing partitions in parallel across blades have to be implemented, as we will describe in Section 5. Furthermore, accessing data via the network incurs higher communication costs than blade-local data access. Finally, different data partitioning strategies have an impact on query performance and load balancing. Therefore, from time to time, it can become necessary to reorganize the partitions to achieve better load balancing or to adapt to a particular query workload. Some of our considerations on data placement and dynamic cluster reorganization can be found in [SEJ⁺11].

After deciding on a multi-blade system as the target hardware, the next question is: should many less powerful low-end blades be used or do we design for a small number of more powerful high-end blades? For SanssouciDB, we chose the latter option, since high-end blades are more reliable and allow more blade-local data processing thereby avoiding expensive network communication to access data on remote blades. In our target hardware configuration, a typical blade contains up to 2 TB of main memory and up to 64 cores. With 25 of these blades, we can manage the enterprise data of the largest companies in the world.

To make efficient use of this architecture, SanssouciDB exploits parallelism at all levels. This includes distributed query processing (among blades), parallel query processing algorithms (among cores on a blade) and exploiting Single Instruction Multiple Data (SIMD) instructions at processor level [WPB⁺09].

Combined columns as shown in Figure 1 are column groups in the sense of fine-grained hybrid data layout [GKP⁺11], which will be discussed in Section 3.1. Column grouping is particularly advantageous for columns that often occur together as join or group-by columns (see also the aggregation and join algorithms presented in Section 5). In the following sections, we will examine the concepts shown in Figure 1 in greater detail.

3 Data Access and Compression

In this section, we describe how SanssouciDB organizes data access in main memory and how compression is used to speed up processing and make efficient use of the available main memory capacity.

3.1 Organizing and Accessing Data in Main Memory

Traditionally, the data values in a database are stored in a row-oriented fashion, with complete tuples stored in adjacent blocks on disk or in main memory. This allows for fast access of single tuples, but is not well suited for accessing a set of values from a single column. The left part of Figure 2 exemplifies this by illustrating the access patterns of two SQL statements: the result of the upper statement is a single tuple, which leads to a sequential read operation of an entire row in the row store. However, accessing a set of attributes leads to a number of costly random access operations as shown in the lower left part. The grey shaded part of the memory region illustrates data that is read, but not required. This happens as data is read from main memory in chunks of the size of a cash line which can be larger than the size of a single attribute.

An analysis of database accesses in enterprise applications has shown that set-based reads are the most common operation [Pla09], making row-oriented databases a poor choice for these types of applications. Column-oriented databases [ABH09], in contrast, are well suited for these set-based operations. In particular, column scans, where all the data values that must be scanned are read sequentially, can be implemented very effi-

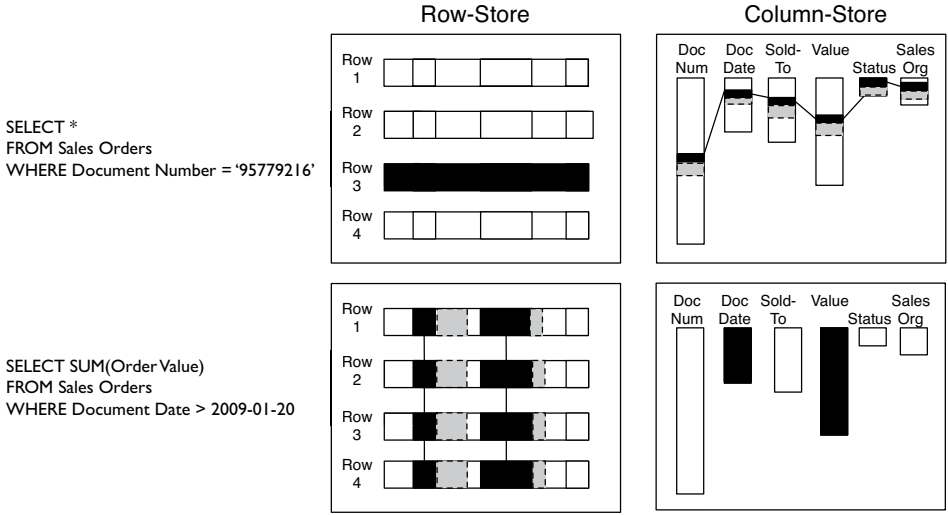
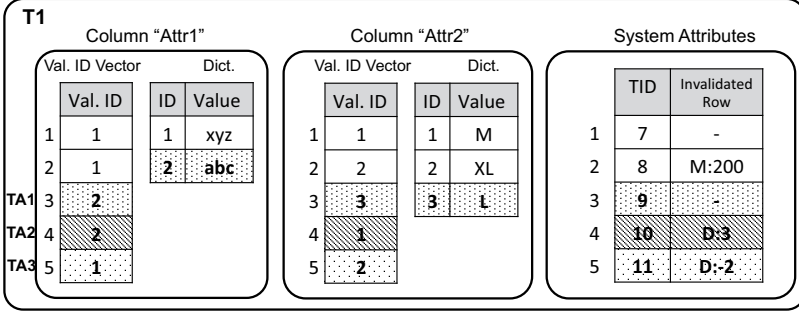


Figure 2: Operations on the Row Store and the Column Store

ciently. The right part of Figure 2 illustrate these considerations. The different lengths of the columns illustrates a varying compression rate; compression is described later in this section. Good scan performance makes column stores a good choice for analytical processing; indeed, many commercial column-oriented databases target the analytics market, for example, SAP Business Warehouse Accelerator and Sybase IQ. The disadvantage of column-oriented databases is that the performance of row-based operations is typically poor. To combine the best of both worlds, SanssouciDB allows certain columns to be stored together, such as columns that are frequently queried as a group. In the following, we refer to these groups of columns as combined columns (see Figure 1). Allowing these column types combines the advantage of the column-oriented data organization to allow for fast reads with good write performance. Further benefits of these combined columns are described in Section 5.

As outlined above, column stores provide good scan performance. To evaluate a query predicate on a column, for example, finding all occurrences of a certain material number, a column scan is applied. However, when the query predicate has a high selectivity, that is, when only a small number of all rows are returned, scanning results in too much overhead. For columns that are often queried with highly selective predicates, like primary or foreign key columns, SanssouciDB allows the specification of inverted indexes (see Figure 1).

To reduce the need for locking and to allow us to maintain a history of all changes to the database, we adopt an insert-only approach. We define the term “insert-only” as follows: An insert-only database system does not allow applications to perform updates or deletions on physically stored tuples of data. In SanssouciDB, all write operations insert a new tuple into the differential buffer, while the main store is only accessed by read operations. To track the different versions of a tuple, a table in the differential buffer contains two



Write Transactions:

TA1 (TID=9): **INSERT into T1 (Attr1, Attr2) values ('abc', 'L');**
TA2 (TID=10): **UPDATE T1 set Attr2='M' where Attr1='abc';**
TA3 (TID=11): **DELETE FROM T1 where Attr2='XL';**

Figure 3: Column Store Write Operations

system attributes for each record: the *TID* of the transaction that wrote the record and an *invalidated row* field referencing to the row that became invalid by inserting this record, i.e., the previous version of the record. In case of an insert operation this field is left empty. Figure 3 depicts an example of insert and update operations and their effect on the differential buffer, for example, TA₂ updates row 3 and inserts D:3 into the invalidated row field to signal that row 3 of the differential buffer is now invalid and is the successor of the record in row 4.

A consequence of this insert-only approach is that data volumes increase over time. Our objective is to always keep all the relevant data in main memory, but as new data is added over time this becomes increasingly difficult. To ensure low latency access to the most recent data we make use of data aging algorithms to partition data into active data, which is always kept in main memory, and passive data that may be moved to flash-based storage, if necessary. The history store, which is kept in non-volatile storage, is responsible for keeping track of passive data. Keeping the history allows SanssouciDB to execute time-travel queries, which reflect the state of the database at any user-specified point in time.

3.2 Compression

As main memory sizes have grown rapidly, access latency to main memory has become the bottleneck for the execution time of computations: processors are wasting cycles while waiting for data to arrive. This is especially true for databases as described in [ADHW99]. While cache-conscious algorithms are one way to improve performance significantly [ADH02, RR00, RR99], another option is to reduce the amount of data transferred from and to main memory, which can be achieved by compressing [WKHM00]. On the one

hand, compression reduces I/O operations between main memory and CPU registers, on the other hand, it leverages the cache hierarchy more effectively since more information fits into a cache line.

The number of CPU cycles required for compressing and decompressing data and the savings in CPU cycles from shorter memory access time result in increased processor utilization. This increases overall performance as long as the database system is I/O bound. Once compression and decompression become so CPU-intensive that the database application is CPU bound, compression has a negative effect on the overall execution time. Therefore, most column-oriented in-memory databases use light-weight compression techniques that have low CPU overhead [AMF06]. Common light-weight compression techniques are dictionary encoding, run-length encoding, bit-vector encoding, and null suppression.

In SanssouciDB, we compress data using dictionary encoding. In dictionary encoding, all values of a column are replaced by an integer called value ID. The original values are kept in a sorted array called dictionary. The value ID is the position of the value in the dictionary; see Figure 4. Our experiments have shown that run-length encoding on a sorted column incurs the fewest amount of cache misses among these compression techniques. However, applying run-length encoding requires sorting of each column before storing it. In order to reconstruct records correctly, we would have to store the original row ID as well. When reconstructing records, each column must be searched for that ID resulting in linear complexity. As enterprise applications typically operate on tables with up to millions records, we cannot use explicit row IDs for each attribute in SanssouciDB, but keep the order of attributes for each tuple identical throughout all columns for tuple reconstruction. Dictionary encoding allows for this direct offsetting into each column and offers excellent compression rates in an enterprise environment where many values, for example, country names, are repeated. Therefore, dictionary encoding suits our needs best and is our compression technique of choice in SanssouciDB. Read performance is also improved, because many operations can be performed directly on the compressed data. For a more complete discussion of compression, we refer the reader to [LSF09].

3.3 Optimizing write operations

As described in the previous section, data is compressed to utilize memory efficiently. This causes write operations to be expensive, because they would require reorganizing the storage structure and recalculating the compression. Therefore, write operations on the column store do not directly modify compressed data of the so-called *main store*, but all changes go into a separate data structure called *differential buffer* as shown in Figure 4. Both structures are dictionary encoded, while the main store is further compressed using additional compression techniques. While the dictionary of the main store is a sorted array, which defines the mapping of a value to its value ID as the position of that value in the array, the dictionary of the differential buffer is an unsorted array, allowing for fast insertion of new values. The arbitrary order of values in the differential buffer dictionary slows down read operations, since a lookup of a value ID has complexity $O(N)$, or $O(\log N)$ if an index structure, e.g., a B+/CSB+ tree is used for value ID lookup. A growing dif-

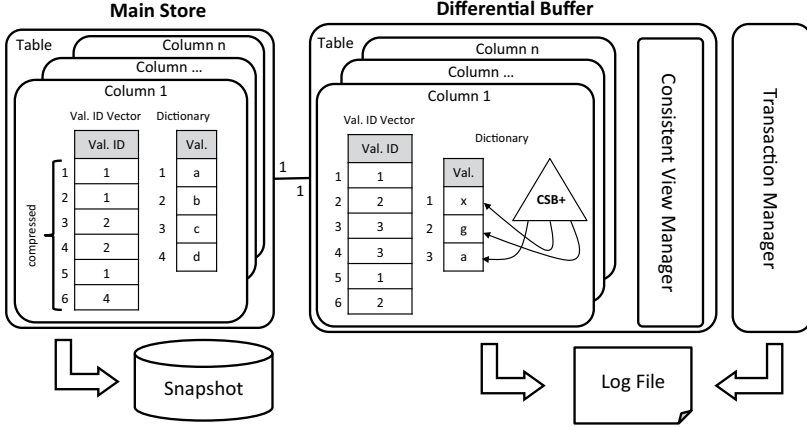


Figure 4: Operations on the Row Store and the Column Store

ferential buffer degrades read performance and increases memory usage since the value ID vector is not further compressed as in the main store and, in addition, the index of the dictionary, i.e., the CSB+ tree, grows fast if a column has many unique values. Therefore, differential buffer and main store are merged from time to time. The merge process is a reorganization of the column store integrating the differential buffer into the main store and clearing the differential buffer afterwards [KGT⁺10].

4 Transaction Management

Enterprise applications require strong transactional guarantees. SanssouciDB uses a Multi Version Concurrency Control (MVCC) scheme to isolate concurrent transactions, while a physiological logging and snapshotting approach is used to persist transactional changes on non-volatile memory to provide fault tolerance. In the following, we give an overview of how the MVCC scheme exploits versioning of tuples provided by the insert-only approach and describe the logging scheme of SanssouciDB.

4.1 Concurrency Control

Isolation of concurrent transactions is enforced by a central transaction manager maintaining information about all write transactions and the *consistent view manager* deciding on visibility of records per table. A so-called *transaction token* is generated by the transaction manager for each transaction and encodes what transactions are open and committed at the point in time the transaction starts. This transaction token is passed to the consistent view manager of each table accessed and is used to emulate the same record visibility as

TID	TA State	CID
...
6	aborted	-
7	committed	7
8	open	-
9	committed	9
10	committed	8
11	committed	10

Table 1: Example for transaction information maintained by the transaction manager.

TID	New Rows	Invalidated Rows
< 8	1, 2	M:20, M:10, M:5
9	3	
10	4	D:3
11		D:2

Table 2: Consistent view information for transactions of Figure 3

at transaction start time.

For token generation, the transaction manager keeps track of the following information for all write transaction: (i) unique transaction IDs (TID), (ii) the state of each transaction, i.e., *open*, *aborted*, or *committed*, and (iii) once the transaction is committed, a commit ID (CID). While CIDs define the commit order, TIDs reflect the start order of transactions. This is exemplified in Table 1. From this information, the transaction manager generates the following values of the transaction token:

- *maxCID*: the highest CID, in our example $\text{maxCID}=10$
- *minWriteTID* and *closedTIDs*: together these values describe all transactions that are closed. In our example we have $\text{minWriteTID}=7$ and $\text{closedTIDs}=\{9,10,11\}$. Changes made by a transaction T_i with $\text{TID}_i < \text{minWriteTID}$ or $\text{TID}_i \in \text{closedTIDs}$ must be visible for transactions T_j if $\text{TID}_j \geq \text{TID}_i$.
- *TID*: for a write transaction this is the unique TID of the transaction. For read transactions this is the TID that would be assigned to the next starting write transaction. Hence, different read transactions might have the same TID. A read transaction T_i is not allowed to see changes made by a transaction T_j , if $\text{TID}_j \geq \text{TID}_i$. If a read transaction is promoted to a write transaction it may be necessary to update the transaction token with a new TID, because the original value might be used by another transaction.

All write operations insert a new record into the column store as part of the insert-only concept (c.f. Section 3.1). Rows inserted into the differential store by an open transaction are not visible to any concurrent transaction. New and invalidated rows are announced to the consistent view manager as soon as the transaction commits. The consistent view manager keeps track of all added and invalidated rows to determine the visibility of records for a transaction.

For every transaction T_i , the consistent view manager maintains two lists: one list with the rows added by T_i and a second list of row IDs invalidated by T_i . For a transaction T_j , changes made by transactions with a TID smaller than T_j 's TID are visible to T_j . For a

compact representation of change information, the consistent view manager consolidates the added row list and invalidated row lists of all transactions with a TID smaller than the *MinReadTID* into a single new row and a single invalidated row list. *MinReadTID* is defined as the maximum TID for which all changes written with the same or a lower TID may be shown to all active transactions. For all transactions T_i with a TID larger than *MinReadTID*, individual change information must be kept. Table 2 depicts the lists maintained by the consistent view manager for our example transactions of Figure 3: TA_1 inserts row 3, while TA_2 invalidates row 3 and adds row 4. The first row of Table 2 shows the consolidation of new and invalidated row lists for all transactions with TIDs smaller than *MinReadTID*=8.

To determine the visibility of tuples for transaction T_i , i.e., with $TID_i=12$, the consistent view manager interprets the transaction token as follows: (i) since for a running transaction T_i the condition $TID_i > \text{MinReadTID}$ holds, all changes listed in the consolidated list (first row in Table 2) are visible for T_i . In addition, all changes made by transactions T_j with $TID_j \leq TID_i$ are visible for T_i . In our example, this are the changes of transactions with TIDs 9, 10, and 11. All changes of T_k with $TID_i \leq TID_k$ are not visible for T_i .

While read operations can access all visible rows without acquiring locks, write transactions are serialized by locks on row level. Besides main store and differential buffer, which store the most recent version of a tuple, each table has a history store containing previous versions of tuples. The history store uses the CID for distinguishing multiple old versions of a tuple. In doing so, SanssouciDB provides a time-travel feature similar to the one described in [Sto87].

4.2 Logging and Recovery

Enterprise applications are required to be resilient to failures. Fault tolerance in a database system refers to its ability to recover from a failure and is achieved by executing a recovery protocol when restarting a crashed database, thereby restoring its latest consistent state before the failure. This state must be derived using log data that survived the failure, in the form of logs residing on a non-volatile medium. Writing log information to non-volatile memory is a potential bottleneck because it is bound by disk throughput. To prevent log writes from delaying transaction processing, SanssouciDB uses a parallel logging scheme to leverage the throughput of multiple physical log volumes.

To recover a table, the main as well as the differential store must be rebuilt in a consistent fashion. The main part of a table is snapshot to a non-volatile medium when main and differential store are merged (c.f. Section 3.1). In order to fall-back to a consistent state during recovery, redo information for all write operations since the last merge is logged to the *delta log*. The central transaction manager writes a *commit log* file recording the state transitions of all write transactions.

At restart of a crashed database, the main part is recovered using its latest snapshot and the differential buffer is recovered by replaying the delta and commit logs. While recovering the main store from a snapshot is fast, for example when using memory-mapped files on

an SSD, replaying log information is a potential bottleneck for fast recovery.

As described in Section 3.1, the main store of a table as well as the differential buffer consists of a vector holding value IDs and a dictionary defining the mapping of value IDs to values. During normal operations, the dictionary builds up over time, i.e., a new mapping entry is created in the dictionary each time a new unique value is inserted. Both value ID vector and dictionary, must be rebuilt during recovery using the delta log.

To allow parallel recovery, we use a physiological logging scheme [GR93] that stores the insert position (row ID) and the value ID for each write operation, called a *value log*. The logging scheme also includes logs for the dictionary mapping (dictionary logs). A value log entry contains the TID, the affected attribute, the row ID, and the value ID of the inserted value defined by the dictionary mapping. A dictionary log contains the attribute, the value, and the value ID.

The system attributes of each record (TID and invalidated row) are logged in the value log format. Combined with commit logs written by the transaction manager, they are used to rebuild the added row and invalidated row lists of the transaction manager at recovery time. A commit log entry contains the TID, the transaction state, and a commit ID if the state is committed.

To reduce the number of log entries for a row, only value logs for attributes that were actually changed by an update operation are persisted. During recovery, the missing attribute values of a record, i.e., attribute values that are not updated but have the same value as the previous version of the record can be derived from the previous version of the record.

Dictionary entries are visible to other transactions before the writing transaction has committed, to prevent the dictionary from becoming a bottleneck during transaction processing. Therefore, dictionary entries must be managed outside the transactional context of a running transaction to prevent removal of a dictionary entry in case of a rollback. For example, if transaction T_i inserts a dictionary mapping 'abc' \rightarrow 1 and is aborted after another transaction T_j reads and uses the mapping for 'abc' from the dictionary, removing the mapping from the dictionary during rollback of T_i would affect T_j , which must not happen. Logging dictionary changes outside the transactional context causes the problem that in case transactions are aborted a dictionary might contain unused values. However, unused entries are removed with the next merge of differential and main store and will not find their way into the new main store.

To speed up recovery of the differential buffer further, the value ID vector as well as the dictionary can be snapshot from time to time, which allows for truncating the delta log.

5 Parallel Aggregation and Join

SanssouciDB runs on a blade architecture with multiple cores per blade (see Section 2). The system can parallelize algorithms along two dimensions: across blades and within a blade. In the first case, we assume a shared-nothing architecture. Each blade is responsible for a certain partition of the data. In the second case, compute threads that run in parallel

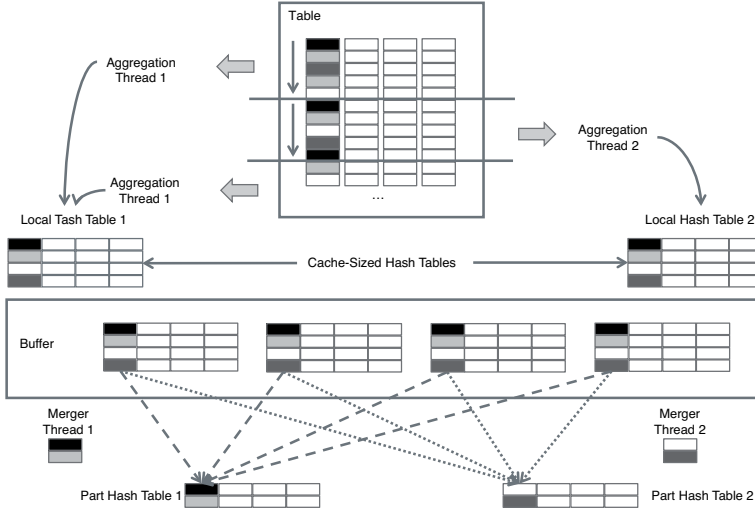


Figure 5: Parallel Aggregation

on one blade can access the same data on that blade, which is typical for a shared-memory architecture. In this section, we want to discuss aggregation and join algorithms developed for SanssouciDB.

5.1 Parallel Aggregation

Aggregation algorithms have been proposed for shared-nothing architectures and for shared-memory architectures. For a discussion on distributed aggregation across blades, we refer the reader to [TLRG08]. Here, we want to discuss the shared-memory variant is implemented utilizing multiple concurrently running threads. We first consider how the input data is accessed (see upper part of Figure 5). Let us assume, we can run n threads in parallel. Initially, we run n aggregation threads. Each aggregation thread (1) fetches a certain rather small partition of the input relation, (2) aggregates this partition, and (3) returns to step (1) until the complete input relation is processed. This approach avoids the problem of unbalanced computation costs: If we would statically partition the table into n chunks, one per thread, computation costs could differ substantially per chunk and thread. Using smaller chunks and dynamically assigning them to threads evenly distributes the computation costs over all threads.

Each thread has a private, cache-sized hash table, into which it writes its aggregation results to. Because the hash table is of the size of the cache, the number of cache misses are reduced. If the number of entries in a hash table exceeds a threshold, for example, when 80 % of all entries are occupied, the aggregation thread initializes a new hash table and moves the old one into a shared buffer. When the aggregation threads are finished, the buffered hash tables have to be merged. This is accomplished by merger threads. The

buffered tables are merged using range partitioning. Each merger thread is responsible for a certain range, indicated by a different shade of grey. The merger threads aggregate their partition into so-called part hash table. Part hash tables are also private to each thread. The partitioning criterion can be defined on the keys of the local hash tables. For example, all keys, whose hashed binary representation starts with an 11, belong to the same range and are assigned to a certain merger thread. The final result can be obtained by concatenating the part hash tables (since they contain disjoint partitions).

The algorithm can be improved by allowing merger threads to run when the input table has not been completely consumed by the aggregation threads. Aggregation phases and merge phases alternate. This allows us to restrict the number of local hash tables in the buffer, thus reducing the memory footprint of the algorithm. Note, a lock is required to synchronize the partitioning operations on the input table and on the hash tables in the buffer. These locks are, however, only held for a very short period of time, because partition ranges can be computed quickly.

5.2 Parallel Join

Similar to the aggregation algorithm, SanssouciDB can compute joins across blades and within one blade utilizing multiple threads. To compute a distributed join across blades, SanssouciDB applies the well-known semijoin method: Let us assume a join between tables R and S on the set of join columns A . Tables R and S are stored on different blades (1 and 2). First, projection $\pi_A(R)$ is calculated on Blade 1. The projection retains the set of join columns A and applies duplicate removal. The projected result is then sent to Blade 2, where the intermediate result $T = \pi_A(R) \bowtie S$ is computed. T is the set of tuples of S that have a match in $R \bowtie S$. Projection $\pi_A(T)$ is sent back to the node storing R to calculate $U = R \bowtie \pi_A(T)$. The final join result is obtained by calculating $U \bowtie T$.

In the distributed join computation, some operations run locally on a blade. For these operations, parallel algorithms tailored to shared-memory architectures can be applied. Because the computation of distinct values and aggregation are closely related, projection $\pi_A(R)$ on Blade 1 can be computed using a slightly modified version of the parallel aggregation algorithm above. Furthermore, the semijoins can also be computed locally in parallel. In the following, we want to sketch a parallel algorithm for join computation.

Just like aggregation, joins can be computed based on hash tables. The algorithm is depicted in Figure 6. The columns with the dashed lines contain row numbers. Row numbers are not physically stored in SanssouciDB. They are calculated on the fly when required. In the preparation phase depicted at the left hand side, the values of the smaller input relation's join attributes and their corresponding row numbers are inserted into a hash table. This insertion is executed in parallel by concurrently running join threads. Again, each thread has a private cache-sized hash table that is placed into the buffer when a certain threshold is reached. The buffer is occasionally merged into part hash tables. The result consists of a set of part hash tables with key-list pairs. The pair's key corresponds to a value from a join column, and the pair's list is the list of row numbers indicating the posi-

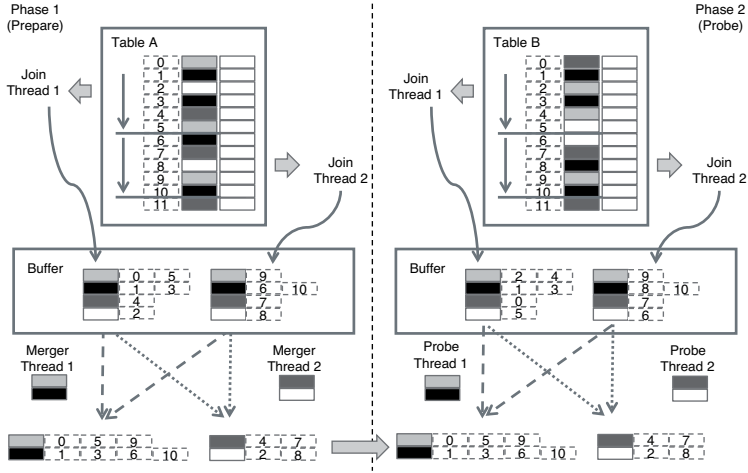


Figure 6: Parallel Join

tion of the value's occurrence in the input relation. Again, the keys in the part hash tables are disjoint.

In the second phase, the algorithm probes the join columns of the larger input relation against the part hash tables. Probing works as follows:

1. Private cache-sized hash tables for the larger input relation are created and populated by a number of concurrently running threads as described above. When a hash table reaches a certain threshold, it is placed in the buffer.
2. When the buffer is full, the join threads sleep and probing threads are notified. In Figure 6, we copied the resulting part hash tables from the preparation phase for simplicity. Each probing thread is responsible for a certain partition. Because the same partitioning criterion is used during the first phase of the algorithm, all join candidates are located in a certain part hash table. If the probing thread finds a certain value in a part hash table, it combines the rows and appends them to the output table.
3. When all hash tables in the buffer have been processed, the join threads are notified.
4. The algorithm terminates when the buffer is empty and the larger input relation has been consumed. The result is the joined table.

This algorithm can be improved by materializing the join result as late as possible. For example, the result of the join can be a virtual table, that contains references to the original tables by keeping a list of row pairs. Furthermore, the join algorithm has to be aware of the columnar data layout. This is also true for the aggregation algorithm. As described so far, when processing a partition, an aggregation or join thread reads values from the participating columns row by row. Row-wise access is expensive in a column-oriented database

system, because row-wise access provokes many cache misses. To remedy this situation, we can use a special hash-table implementation that allows column-wise insertion. The details are however beyond the scope of this paper.

5.3 Business Analytics on Transactional Data

Good aggregation performance is one of the main objectives in data warehouses. One approach to achieve the necessary performance in disk-based relational data warehouses is to consolidate data into cubes, typically modeled as a star or snowflake schemas. In a cube, expensive joins and data transformations are pre-computed and aggregation is reduced to a mere scan across the central relational fact table. Due to the possibly large size of fact tables in real-world scenarios (a billion rows and more are not unusual), further optimizations such as materialized aggregates, became necessary.

With the introduced aggregation and join algorithms that exploit modern hardware, redundant data storage using cubes and materialized aggregates are not necessary anymore. Executing business analytics on transactional data that is stored in its original, normalized form becomes possible with SanssouciDB. Avoiding materialized aggregates and the need for the star schema dramatically reduces system complexity and TCO, since cube maintenance becomes obsolete. Calculations defined in the transformation part of the former ETL process are moved to query execution time. Figure 7 shows an example of an architecture that uses views that can be stacked in multiple layers. Any application on the presentation layer can access data from the virtually unified data store using views. Views store no data but only transformation descriptions such as computations, possibly retrieving data from several sources, which might again be views. Virtual cubes are similar to views but provide the same interface as real cubes.

Our architecture enables the instant availability of the entire data set for flexible reporting. Moreover, this approach fulfills many of the ideas mentioned in the context of BI 2.0 discussions, where BI is “becoming proactive, real-time, operational, integrated with business processes, and extends beyond the boundaries of the organization” [Rad07]. In the next section, we will showcase some sample applications that make use of our new architecture.

6 Application Examples

We have extensively used SanssouciDB in enterprise application prototypes to leverage its technological potential. In the following, we want to discuss our findings by showing how SanssouciDB can be used to implement the dunning and the availability-to-promise applications.

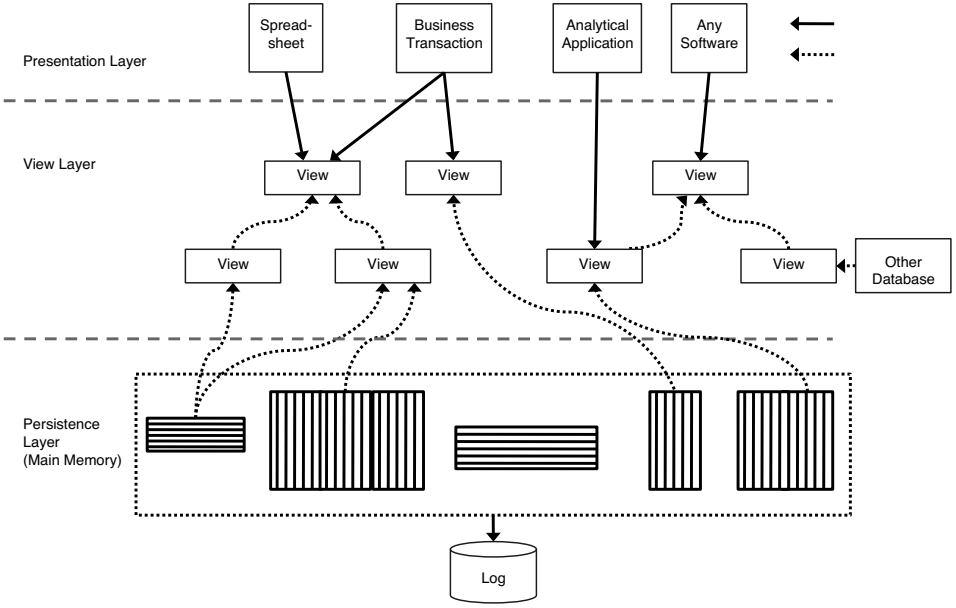


Figure 7: The View Layer Concept

6.1 Dunning

The dunning business process is one of the most cost-intensive business operations. Dunning computes the balance between incoming payments and open invoices for all customers. Depending on the amount and the delay of an outstanding payment, different dunning levels are assigned to customers. Depending on the dunning level, pre-defined actions are triggered, e.g., service blacklisting. Performing dunning runs in current enterprise systems has an impact on OLTP response times, because of the large number of invoices and customers accounts that must be scanned. To reduce the impact on the response time of the operational system, dunning runs are typically performed at the night or over the weekend. Our field study at a German mobile phone operator revealed that dunning runs in current enterprise applications can only be performed at daily or weekly intervals for all customers.

The dunning process is mainly a scan of a long table containing millions of entries. In a prototype, we adapted dunning algorithms that were formerly implemented in the application layer and re-implemented them using stored procedures in our new storage engine. To join invoice data with customer data and to calculate the amounts due, our implementation makes heavy use of the join and aggregation algorithms introduced in Section 5. Our results show that SanssouciDB is able to improve the execution time of the dunning run from more than 20 minutes to less than one second. This outcome shows that in-memory technology is capable of improving the response time of existing applications by orders of magnitude.

As a consequence, the former batch-oriented dunning run could be transformed into an interactive application. For example, a check of the current dunning status over all customers can now be performed instantaneously. This enables managers to query the top ten overdue invoices and their amount on their personal cellular phone, notebook, or any mobile device with an Internet connection at any time.

6.2 Availability-To-Promise

ATP provides a checking mechanism to obtain feasible due dates for a customer order. This is done by comparing the quantities of products which are in stock or scheduled for production against the quantities of products which are assigned to already promised orders. A common technique in current Supply Chain Management systems is to use materialized aggregates to keep track of stock quantities, which results in having a separate aggregate for each product. This means that a new shipment would increase the value of each shipped product's aggregate, while the assignment of products to a confirmed customer order would decrease it. We analyzed an existing ATP implementation and found the following: Although the use of aggregates reduces the necessary amounts of I/O operations and CPU cycles for the single ATP check itself, it introduces the following disadvantages:

- *Redundant Data*: One problem that arises in association with materialized aggregates is the need to maintain them. To preserve a consistent view on the data across the whole system, every write operation has to be propagated to the materialized aggregates. Even if the updates are triggered immediately, they still imply delays causing temporary inconsistencies. Additionally, even if the amount of I/O operations and CPU cycles is reduced to a minimum for the check itself by using aggregates, the overall sum of required operations might be higher due to synchronization as well as maintenance and costly back calculation of the aggregates.
- *Exclusive Locking*: All modifications to an aggregate require exclusive access to the respective database entity and block concurrent read and write processes. The downside of locking is obvious, as it queues the incoming requests and affects the performance significantly in case of a highly parallel workload.
- *Inflexible Data Querying*: Materialized aggregates are tailored to a predefined set of queries. Unforeseeable operations referring to attributes that were not considered at design time cannot be answered with these pre-aggregated quantities. Those attributes include, for example, the shelf life, product quality, customer performance, and other random characteristics of products, orders, or customers. Additionally, due to the use of materialized aggregates, the temporal granularity of the check is fixed. Once the aggregates are defined and created, based on, for example, the available quantities per day, it is not possible to perform ATP checks on an hourly granularity.
- *Inflexible Data Schema Extensions*: The previously mentioned inflexibility of not being able to change the temporal granularity of a single check indicates another

related disadvantage: the inability to change the data schema, once an initial definition has been done. The change of the temporal check granularity or the inclusion of a previously unconsidered attribute is only possible with a cumbersome reorganization of the existing data.

- *No Data History*: Maintaining aggregates instead of recording all transactions enriched with information of interest means to lose track of how the aggregates have been modified. In other words, no history information is available for analytics or for rescheduling processes.

As we have outlined in Section 5.3, SanssouciDB can execute business analytics on operational data. No star schema or materialized aggregates are required. To leverage this advantage for our ATP implementation, we developed a prototype that was designed with the following aspects in mind: Pre-aggregated totals were completely omitted by storing stock data at the finest granularity level in SanssouciDB. For an ATP check, this up-to-date list is scanned. Removing the materialized aggregates allowed us to implement a lock-free ATP check algorithm. Eliminating locks has the advantage of concurrent data processing, which works especially well in hotspot scenarios, when multiple users perform ATP checks concurrently.

For the prototype implementation, we used anonymized customer data from a Fortune 500 company with 64 million line items per year and 300 ATP checks per minute. Analysis of the customer data showed that more than 80 % of all ATP checks touch no more than 30 % of all products. A response time of 0.7 s per check was achieved using serial checking. In contrast, we were able to execute ten concurrently running checks with our adapted reservation algorithm in a response time of less than 0.02 s per check.

Our approach eliminates the need for aggregate tables, which reduces the total storage demands and the number of inserts and updates required to keep the totals up to date. Because insert load in SanssouciDB could be reduced, the spare capacity can be used by queries and by instant aggregations. Since all object instances are traversed for aggregation, fine-grained checks on any product attribute are possible. The old ATP implementation was kept lean by focusing on a minimum number of attributes per product. For example, printers of a certain model with a German and an English cable set are grouped together or both cable sets are added to improve performance. With the help of SanssouciDB, fine product attributes could be managed in a fine-grained manner. Thus, printers including different cable sets could be considered as individual products during ATP checks.

7 Conclusion

With SanssouciDB, we have presented a concept that we believe is the ideal in-memory data management engine for the next generation of real-time enterprise applications. Its technological components have been prototypically implemented and tested individually in our research. To conclude this paper, we present a few cases where the application of the concepts of SanssouciDB has already led to promising results.

SAP's latest solution for analytical in-memory data management called HANA (High-Performance Analytical Appliance) uses many of the concepts of SanssouciDB, and is currently being rolled out to pilot customers in different industries. HANA's performance impact on common reporting scenarios is stunning. For example, the query execution times on 33 million customer records of a large financial service provider dropped from 45 minutes on a traditional DBMS to three seconds on HANA. The speed increase fundamentally changes the company's opportunities for customer relationship management, promotion planning, and cross selling. Where once the traditional data warehouse infrastructure has been set up and managed by the IT department to pre-aggregate customer data on a monthly basis, HANA now enables end users to run live reports directly against the operational data and to receive the results in seconds.

In a similar use case, a large vendor in the construction industry is using HANA to analyze its nine million customer records and to create contact listings for specific regions, sales organizations, and branches. Customer contact listing is currently an IT process that may take two to three days to complete. A request must be sent to the IT department who must plan a background job that may take 30 minutes and the results must be returned to the requestor. With HANA, sales people can directly query the live system and create customer listings in any format they wish, in less than 10 s.

A global producer of consumer goods was facing limitations with its current analytical system in that it was not able to have brand and customer drilldowns in the same report, which is not possible since joins are no longer pre-computed. Finally, the query execution times for a profitability analysis application of the same customer were reduced from initially ten minutes to less than ten seconds.

References

- [ABH09] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column oriented Database Systems. *PVLDB*, 2(2):1664–1665, 2009.
- [ADH02] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.
- [ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277, 1999.
- [AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682, 2006.
- [GKP⁺11] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE - A Hybrid Main Memory Storage Engine. *PVLDB*, 2011.
- [GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

- [KGT⁺10] Jens Krüger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Franz Faerber. Optimizing Write Performance for Read Optimized Databases. In *Database Systems for Advanced Applications, 15th International Conference, DAS-FAA 2010, Tsukuba, Japan, April 1-4, 2010, Proceedings, Part II*, pages 291–305, 2010.
- [LSF09] Christian Lemke, Kai-Uwe Sattler, and Franz Färber. Kompressionstechniken für spaltenorientierte BI-Accelerator-Lösungen. In *Datenbanksysteme in Business, Technologie und Web (BTW 2009), 13. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, 2.-6. März 2009, Münster, Germany*, pages 486–497, 2009.
- [Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1–2, 2009.
- [Rad07] N. Raden. Business Intelligence 2.0: Simpler, More Accessible, Inevitable, 2007. <http://intelligent-enterprise.com>. Retrieved January 14th 2011.
- [RR99] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 78–89, 1999.
- [RR00] Jun Rao and Kenneth A. Ross. Making B⁺-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 475–486, 2000.
- [SEJ⁺11] Jan Schaffner, Benjamin Eckart, Dean Jacobs, Christian Schwarz, Hasso Plattner, and Alexander Zeier. Predicting In-Memory Database Performance for Automating Cluster Management Tasks. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16 2011, Hannover, Germany*. IEEE, Apr 2011. (to appear).
- [SMA⁺07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1150–1160, 2007.
- [Sto87] Michael Stonebraker. The Design of the POSTGRES Storage System. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 289–300, 1987.
- [TLRG08] David Taniar, Clement H. C. Leung, J. Wenny Rahayu, and Sushant Goel. *High Performance Parallel Database Processing and Grid Databases*. John Wiley & Sons, 2008.
- [WKHM00] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3):55–67, 2000.
- [WPB⁺09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.

The Web as the development platform of the future

Nelson Mattos
Google EMEA
Brandschenkestrasse 110
8002 Zürich, Switzerland
mattos@google.com

Abstract: The pace the world of computing has evolved since mainframes has been amazing, dramatically changing the way we design applications and creating opportunities for new players to enter the market. The Web is no longer just a collection of pages anymore and becomes a vital platform for today's most innovative web applications. The explosion of the use of the Internet from mobile devices and applications require powerful backends: this emphasizes how important designing and deploying cloud infrastructure has become. This talk will advocate that the development platform of the future is the web with its browsers and will give you insights into Google's vision in this space and showcase some exciting applications that leverage those technologies to solve incredible tasks. Hopefully I will be able to inspire you to take advantage of the incredible opportunities at hand!

The Power of Declarative Languages: From Information Extraction to Machine Learning

Shivakumar Vaithyanathan
IBM Research – Almaden
650 Harry Road
San Jose, California 95120-6099
shiv@almaden.ibm.com

Abstract: As advanced analytics has become more mainstream in enterprises, usability and system-managed performance optimizations are critical for its wide adoption. As a result, there is an active interest in the design of declarative languages in several analytics areas. In this talk I will describe the efforts in IBM around three areas namely Information Extraction, Entity Resolution and Machine Learning. I will detail these efforts, at some length, and also explain the motivation behind some of the design choices made while implementing declarative solutions for the individual areas. I will end with results that demonstrate multiple advantages of the declarative approaches as compared with existing solutions.

Wissenschaftliches Programm

MapReduce and PACT - Comparing Data Parallel Programming Models

Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske,
Odej Kao, Volker Markl, Erik Nijkamp, Daniel Warneke

Technische Universität Berlin, Germany
Einsteinufer 17
10587 Berlin, Germany
firstname.lastname@tu-berlin.de

Abstract: Web-Scale Analytical Processing is a much investigated topic in current research. Next to parallel databases, new flavors of parallel data processors have recently emerged. One of the most discussed approaches is MapReduce. MapReduce is highlighted by its programming model: All programs expressed as the second-order functions *map* and *reduce* can be automatically parallelized. Although MapReduce provides a valuable abstraction for parallel programming, it clearly has some deficiencies. These become obvious when considering the tricks one has to play to express more complex tasks in MapReduce, such as operations with multiple inputs.

The Nephele/PACT system uses a programming model that pushes the idea of MapReduce further. It is centered around so called *Parallelization Contracts (PACTs)*, which are in many cases better suited to express complex operations than plain MapReduce. By the virtue of that programming model, the system can also apply a series of optimizations on the data flows before they are executed by the Nephele runtime system.

This paper compares the PACT programming model with MapReduce from the perspective of the programmer, who specifies analytical data processing tasks. We discuss the implementations of several typical analytical operations both with MapReduce and with PACTs, highlighting the key differences in using the two programming models.

1 Introduction

Today's large-scale analytical scenarios face Terabytes or even Petabytes of data. Because many of the early large-scale scenarios stem from the context of the Web, the term *Web-Scale Analytical Processing* has been coined for tasks that transform or analyze such vast amounts of data. In order to handle data sets at this scale, processing tasks run in parallel on large clusters of computers, using their aggregate computational power, main memory, and I/O bandwidth. However, since developing such parallel programs bottom-up is a cumbersome and error-prone task, new programming paradigms which support the automatic parallelization of processing jobs have gained a lot of attention in recent years.

The MapReduce paradigm [DG04] is probably the best-known approach to simplify the development and parallel execution of data processing jobs. Its open-source implementation

Hadoop [Had] is very popular and forms the basis of many parallel algorithms that have been published in the last years ([VCL10, Coh09, Mah]). Compared to parallel relational databases, which have been the predominant solution to parallel data processing, MapReduce proposes a more generic data and execution model. Based on a generic key/value data model, MapReduce allows programmers to define arbitrarily complex user functions which then are wrapped by second-order functions *map* or *reduce*. Both of these second-order functions provide guarantees on how the input data is passed to the parallel instances of the user-defined function at runtime. That way, programmers can rely on the semantics of the second-order functions and are not concerned with the concrete parallelization strategies.

However, even though the two functions *Map* and *Reduce* have proven to be highly expressive, their originally motivating use-cases have been tasks like log-file analysis or web-graph inverting [DG04]. For many more complex operations, as they occur for example in relational queries, data-mining, or graph algorithms, the functions *Map* and *Reduce* are a rather poor match [PPR⁺09]. A typical example is an operation that matches key/value pairs with equal keys from two different inputs. Such an operation is crucial for many tasks, such as relational joins and several graph algorithms [YDHP07]. To express it in MapReduce, a typical approach is to form a union of the inputs, tagging the values such that their originating input is known. The Reduce function separates the values by tag again to re-obtain the different input sets. That is not only an unintuitive procedure, programmer must also make explicit assumptions about the runtime parallelization while writing the user code. In fact, many implementations exploit the fact that MapReduce systems, like Hadoop, implement a static pipeline of the form *split-map-shuffle-sort-reduce*. The shuffle and sort basically exercise a parallel grouping to organize the data according to the Reduce function's requirements. Because all operations are customizable, many tasks are executed with appropriate custom split or shuffle operations. That way, MapReduce systems become a parallel process runtime that execute hard-coded parallel programs [DQRJ⁺10]. This clearly conflicts with the MapReduce's initial design goals. While highly customizing the behavior allows to run more complex programs, it destroys the idea of a declarative specification of parallelism, preventing any form of optimization by the system. Especially the incorporation of runtime adaptation to the parallelization methods (such as broadcasting or partitioning) requires a clear specification of the user function's requirements for parallelization, rather than a hard-coding of the method.

To overcome those shortcomings, we have devised a programming model that offers the same abstraction level as MapReduce but pushes its concepts further. It is centered around so called *Parallelization Contracts* (PACTs), which can be considered a generalization of MapReduce [BEH⁺10]. The PACT programming model eases the expression of many operations and makes it more intuitive. Moreover, its extended expressiveness also enables several optimizations to be applied automatically, resulting in more efficient processing.

In this paper, we give a short introduction to both programming models and compare them from the perspective of the developer writing the data analysis tasks. We present tasks from the domains of relational queries, XQuery, data mining, and graph algorithms and present their implementations using MapReduce and PACT. We discuss the differences of the two programming models and the implications for the programmer. The remainder of the paper

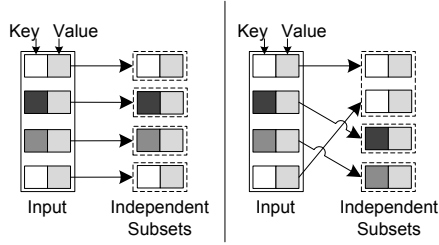


Figure 1: a) Map | b) Reduce

is structured as follows: Section 2 discusses the MapReduce and the PACT programming model in detail, highlighting key differences. Section 3 describes five typical analytical tasks and compares their MapReduce and PACT implementations. Section 4 discusses related work, Section 5 concludes the paper.

2 Parallel Programming Models

This section contrasts the parallel programming models MapReduce and PACT. Since the PACT programming model is a generalization of MapReduce, we start with a short recapitulation of MapReduce before introducing the extensions in the PACT programming model. A more thorough description of the PACT programming model can be found in [BEH⁺10].

2.1 The MapReduce Programming Model

The MapReduce programming model was introduced in 2004 [DG04]. Since then, it has become very popular for large-scale batch data processing. MapReduce founds on the concept of data parallelism. Its data model is key/value pairs, where both keys and values can be arbitrarily complex. A total order over the keys must be defined.

The key idea of MapReduce originates from functional programming and is centered around two second-order functions, *Map* and *Reduce*. Both functions have two input parameters, a set of key/value pairs (input data set) and a user-defined first-order function (user function). *Map* and *Reduce* apply the user function on subsets of their input data set. Thereby, all subsets are independently processed by the user-defined function.

Map and *Reduce* differ in how they generate those subsets from their input data set and pass them to the attached user function:

- *Map* assigns each individual key/value pair of its input data set to an own subset. Therefore, all pairs are independently processed by the user function.
- *Reduce* groups the key/value pairs of its input set by their keys. Each group becomes an individual subset which is then processed once by the user-defined function.

Figure 1 depicts how *Map* and *Reduce* build independently processable subsets. Each subset is processed once by exactly one instance of the user-defined function. The actual functionality of the *Map* and *Reduce* operations is largely determined by their associated user functions. The user function has access to the provided subset of the input data and can be arbitrarily complex. User functions produce none, one, or multiple key/value pairs. The type of the produced keys and values may be different from those of the input pairs. The output data set of *Map* and *Reduce* is the union (without eliminating duplicates) of the results of all evaluations of the user function.

A MapReduce program basically consists of two stages which are always executed in a fixed order: In the first stage the input data is fed into the *Map* function that hands each key/value pair independently to its associated user function. The output of the *Map* function is repartitioned and then sorted by the keys, such that each group of key/value pairs with identical keys can be passed on to the *Reduce* function in the second stage. The user function attached to the *Reduce* can then access and process each group separately. The output of the *Reduce* function is the final output of the MapReduce program. As complex data processing tasks do often not fit into a single MapReduce program, many tasks are implemented using a series of consecutive MapReduce programs.

Since all invocations of the user functions are independent from each other, processing jobs written as MapReduce programs can be executed in a massively parallel fashion. In theory *Map* can be parallelized up to the number of input key/value pairs. *Reduce*'s maximum degree of parallelism depends on the number of distinct keys emitted in the map stage.

A prominent representative of a MapReduce execution engine is Apache's Hadoop [Had]. This paper focuses on the abstraction to write parallelizable programs. Therefore, we do not discuss the execution of MapReduce programs. Details can be found in [DG04].

2.2 The PACT Programming Model

The PACT programming model [BEH⁺10, ABE⁺10] is a generalization of MapReduce [DG04] and is also based on a key/value data model. The key concept of the PACT programming model are so-called *Parallelization Contracts (PACTs)*. A PACT consists of exactly one second-order function which is called *Input Contract* and an optional *Output Contract*. Figure 2 visualizes the aspects of a PACT. An Input Contract takes a first-order function with task-specific user code and one or more data sets as input parameters. The Input Contract invokes its associated first-order function with independent subsets of its

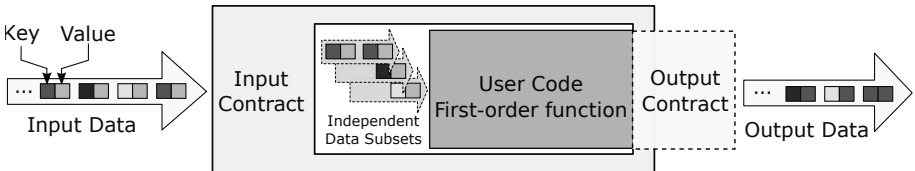


Figure 2: Parallelization Contract (PACT)

input data in a data-parallel fashion. In the context of the PACT programming model, MapReduce’s *Map* and *Reduce* functions are Input Contracts. The PACT programming model provides additional Input Contracts that complement *Map* and *Reduce*, of which we will use the following three in this paper:

- *Cross* operates on multiple inputs of key/value pairs and builds a Cartesian product over its input sets. Each element of the Cartesian product becomes an independent subset.
- *CoGroup* groups each of its multiple inputs along the key. Independent subsets are built by combining the groups with equal keys of all inputs. Hence, the key/value pairs of all inputs with the same key are assigned to the same subset.
- *Match* operates on multiple inputs. It matches key/value pairs from its input data sets with the same key. Each possible two key/value pairs with equal key form an independent subset. Hence, two pairs of key/value pairs with the same key are processed independently by possibly different instances of the user function, while the CoGroup contract assigns them to the same subset and guarantees to process them together.

Figure 3 illustrates how *Cross*, *Match*, and *CoGroup* build independently processable subsets.

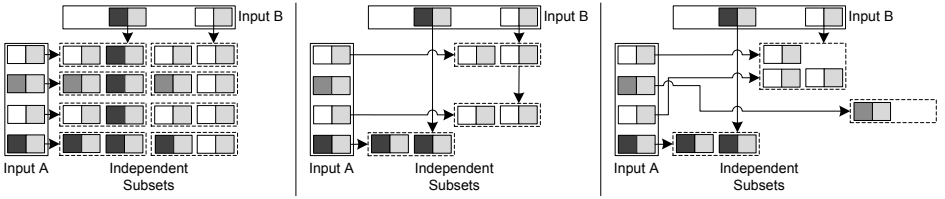


Figure 3: a) Cross | b) Match | c) CoGroup

In contrast to Input Contracts, which are a mandatory component of each PACT, Output Contracts are optional and have no semantic impact on the result. Output contracts give hints about the behavior of the user code. To be more specific, they assert certain properties of a PACT’s output data. An example of such an output contract is the *SameKey* contract. When attached to a Map function, it states that the user code will not alter the key, i.e. the type and value of the key remain after the user code’s invocation and are the same in the output as in the input. Those hints can be exploited by an optimizer that generates parallel execution plans. The aforementioned SameKey contract can frequently help to avoid unnecessary repartitioning and therefore expensive data shipping. Hence, Output Contracts can significantly improve the runtime of a PACT program. Currently, developers must manually annotate user functions with Output Contracts. However, automatic derivation based on static code analysis, or suggestions inferred from runtime observations, are worth exploring.

In contrast to MapReduce, multiple PACTs can be arbitrarily combined to form more complex data processing programs. Since some PACTs naturally expect multiple input data

sets, the resulting data processing program is not necessarily a strict pipeline like in the MapReduce case but can yield arbitrarily complex data flow graphs. In order to implement the program, each PACT in the data flow graph must be provided with custom code (the user function). Furthermore, at least one data source and data sink must be specified.

2.3 Comparing MapReduce and PACT

For many complex analytical tasks, the mapping to MapReduce programs is not straightforward and requires working around several shortcomings of the programming model. Such shortcomings are for example the limitation to only one input, the restrictive set of two primitive functions *Map* and *Reduce*, and their strict order. The workarounds include the usage of auxiliary structures, such as the distributed cache, and custom partitioning functions. However, the necessity to apply such tricks destroys the desired property of transparent parallelization.

The PACT programming model has been explicitly designed to overcome the problems with more complicated analytical tasks. It is based on the concept of Input Contracts, which are a generalizations of the Map and Reduce functions. Compared to MapReduce, it offers several additions: First, it offers a richer set of parallelization primitives (which also include Map and Reduce). The generic definition of Input Contracts allows to extend this set with more special contracts. Second, with the concept of Output Contracts it is possible to declare certain properties of the user functions to improve efficiency of the task's execution. Thirdly, PACTs can be freely assembled to data flows, in contrast to MapReduce's fixed order of *Map* and *Reduce*. Hence, PACT avoids identity mapper or reducer which are frequently required within MapReduce implementation. All those features of the PACT programming model significantly ease the implementation of many complex data processing tasks, compared to the MapReduce approach.

MapReduce programs are always executed with a fixed strategy. The input data is read from a distributed filesystem and fed to the Map function. The framework repartitions and sorts the output of the Map function by the key, groups equal keys together and calls the Reduce function on each group. Due to the declarative character of Input Contracts, PACT programs can have multiple physical execution plans with varying performance. For example, the definition of the Match contract is such that a parallel execution of the attached user function can choose among the following strategies: 1) repartition and sort both inputs by the key (in the same way as MapReduce), or 2) broadcast one of the inputs to all instances and not transferring data from the other input between parallel instances. The choice of the execution strategy is made by the PACT compiler, which translates PACT programs to parallel schedules for the Nephele runtime. The compiler uses an optimizer in a similar fashion as a relational database and selects the strategies that minimize the data shipping for the program. We refer to [BEH⁺10] for details.

3 Comparing Common Task Implementation

In this section we present a selection of common data analysis tasks. The tasks are taken from the domains of relational OLAP queries, XQuery, data mining, and graph analysis algorithms. For each task we give a detailed description and implementations for both programming models, MapReduce and PACTs, pointing out key differences and their implications. All of the presented PACT implementations were designed manually. While Nephele/PACT takes away much work from the programmer, the actual process of defining a given problem as a PACT program still has to be done manually. The automatic generation of PACT programs from declarative languages like SQL or XQuery is a field of research.

3.1 Relational OLAP Query

Task Even though relational data is traditionally analyzed through parallel relational database management systems, there is increasing interest to process it with MapReduce and related technologies [TSJ⁺09]. Fast growing data sets of semi-structured data (e.g. click-logs or web crawls) are often stored directly on file systems rather than inside a database. Many enterprises have found an advantage in a central cheap storage that is accessed by all of their applications alike. For the subsequent analytical processing, declarative SQL queries can be formulated as equivalent MapReduce or PACT programs. Consider the SQL query below. The underlying relational schema was proposed in [PPR⁺09] and has three tables: Web-page documents, page rankings, and page-visit records. The query selects the documents from the relation Documents d containing a set of keywords and joins them with Rankings r , keeping only documents where the rank is above a certain threshold. Finally, the anti-join (the *not exists* subquery) reduces the result set to the documents not visited at the current date.

```
1:  SELECT *
2:    FROM Documents d JOIN Rankings r
3:      ON r.url = d.url
4:  WHERE CONTAINS(d.text, [keywords])
5:    AND r.rank > [rank]
6:    AND NOT EXISTS
7:      (SELECT * FROM Visits v
8:       WHERE v.url = d.url AND v.visitDate = CURDATE());
```

MapReduce To express the query in MapReduce, we intuitively need two successive jobs, which are shown on the left-hand side of Figure 4: The first MapReduce job performs an inner-join (lines 1-5 of the sample query), the second one an anti-join (lines 6-8). The first Map task processes the input relations Documents d , Rankings r and carries out the specific selection (line 4-5) based on the source relation of the tuple. To associate a tuple with its source relation, the resulting value is augmented with a lineage tag. The subsequent reducer

collects all tuples with equal key, forms two sets of tuples based on the lineage tag and forwards all tuples from r only if a tuple from d is present. The second mapper acts as an identity function on the joined intermediate result j and as a selection on the relation v . Finally, the second reducer realizes the anti-join by only emitting tuples (augmented with a lineage tag ' j ') when no v tuple (augmented with a lineage tag ' v ') with an equal key is found.

The implementation can be refined to a single MapReduce job. Since the key is the same for the join and anti-join, both operations can be done together in a single reduce operation. As before, the mapper forms a tagged union of all three inputs and the reducer separates the values for each key, based on their lineage. The reduce function concatenates the values from the d and r , if no value from v is found. The advantage of this implementation is that all inputs are repartitioned (and hence transferred over the network) only once.

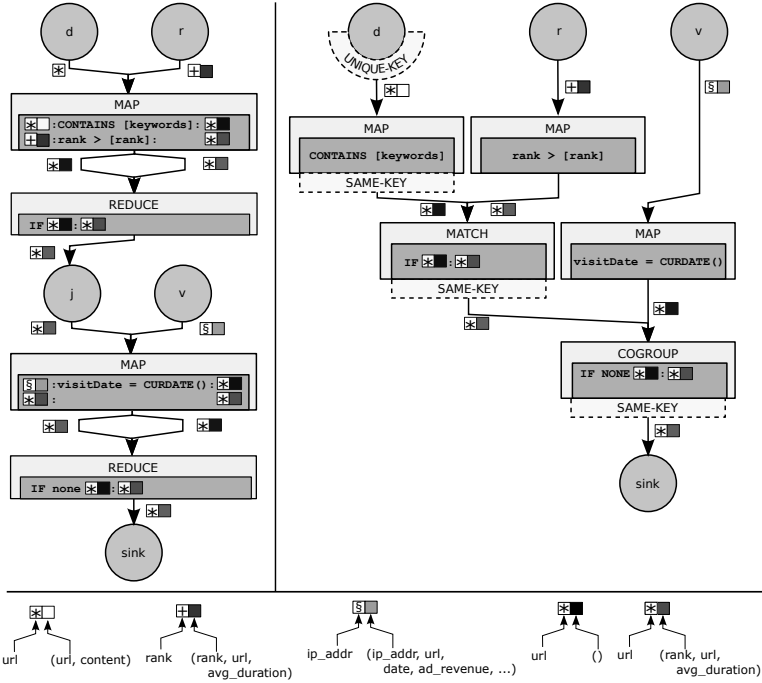


Figure 4: OLAP query as MapReduce (left) and PACT program (right).

PACT The corresponding PACT program for the query is illustrated on the right-hand side of Figure 4. The selections on the relations d , r , v are implemented by three separate user functions attached to the Map contract. The first Reduce task of the original MapReduce implementation is replaced by a Match task. Since the Match contract already guarantees to associate all key/value pairs with the same key from the different inputs pairwise together, the user function only needs to concatenate them to realize the join. The SameKey output

contract is attached to the Match task, because the URL, which has been the key in the input, is still the key in the output. Ultimately, the CoGroup task realizes the anti-join by emitting the tuples coming from the Match task, if for the current invocation of the CoGroup’s user function the input of selected tuples from relation v is empty.

With the *SameKey* output contract attached to the Match, the compiler infers that any partitioning on the Match’s input exists also in its output¹. If the Match is parallelized by partitioning the inputs on the key, the system reuses that partitioning for the CoGroup and only partitions the *visits* input there. That yields the same efficient execution as the hand-tuned MapReduce variant that uses only a single job. However, the PACT implementation retains the flexibility to parallelize the Match differently in case of varying input sizes. In future versions, even dynamic adaption at runtime based on observed behavior of the user-code might be possible.

3.2 XQuery

Task XML is a popular semi-structured data model. It has been adopted by many web-related standards (e.g. XHTML, SVG, SOAP) and is widely used to exchange data between integrated business applications. Because XML is self-describing, it is also a frequent choice for storing data in a format that is robust to application changes. XML data is typically analyzed by evaluating XQuery statements. As the volume of the data stored in XML increases, so does the need for parallel XQuery processors.

The following example shows an XML file describing employees and departments. The XQuery on the right-hand side finds departments with above-average employee count and computes their average salaries.

<pre> <company> <departments> <dep id="D1" name="HR" active="true"/> [...] </departments> <employees> <emp id="E1" name="Kathy" salary="2000" dep="D1"/> [...] </employees> </company> </pre>	<pre> let \$avgcnt := avg(for \$d in //departments/dep let \$e := //employees/emp[@dep=\$d//@id] return count(\$e)) for \$d in //departments/dep let \$e :=//employees/emp[@dep=\$d//@id] where count(\$e)>\$avgcnt and data(\$d//@active)="true" return <department>{ <name>{data(\$d//@name)}</name>, <avgSal>{avg(\$e//@salary)}</avgSal> }</department> </pre>
---	---

a) XML data excerpt | b) XQuery example

¹Omitting Match’s output contract prevents reusing its partitioning. If the Match is parallelized through the partitioning strategy, the execution is similar to that of the original MapReduce implementation with two jobs.

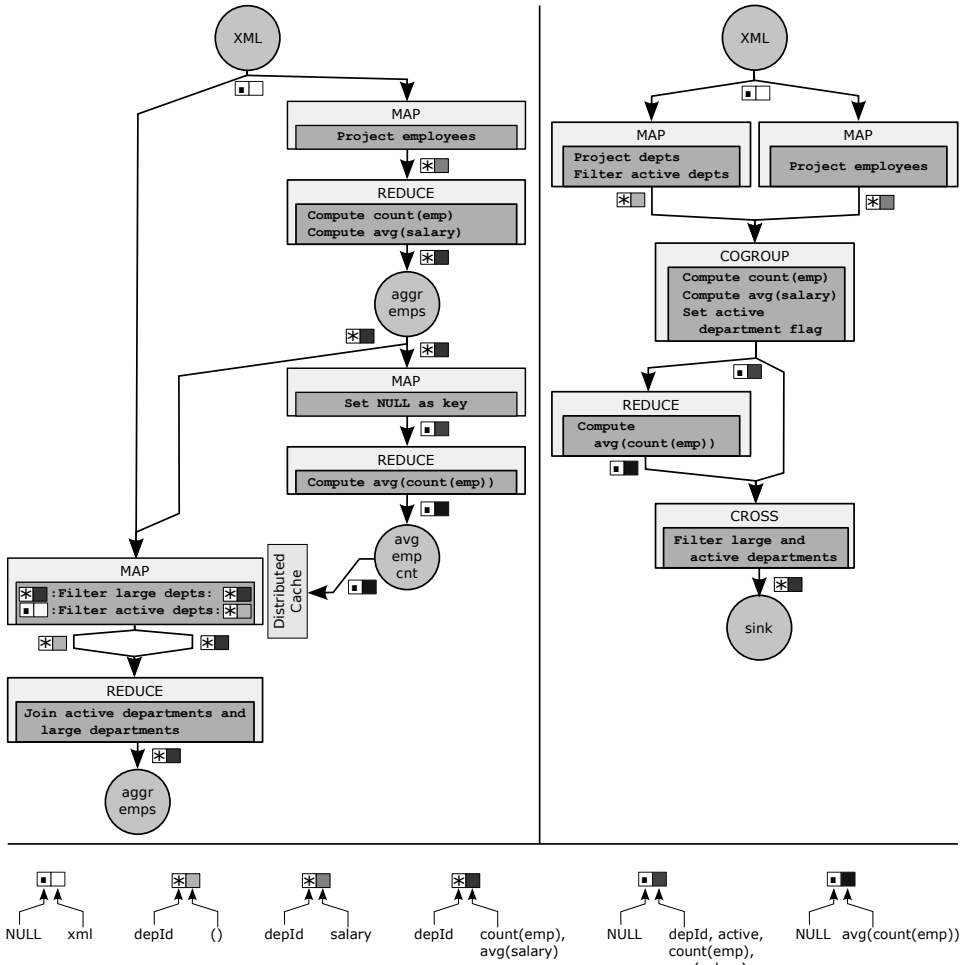


Figure 5: XQuery implementation for MapReduce (left) and PACT (right).

MapReduce Implementing the above XQuery requires in total three MapReduce jobs, as depicted on the left-hand side of Figure 5. The individual jobs are described in the following:

- The first MapReduce job (top) projects employee tuples from the XML file and aggregates the number of employees and their average salary per department. While the mapper performs the projection, the reducer consumes the employee tuples per department and takes care of the aggregations.
- The second job (middle) computes the average number of employees per department. The mapper consumes the result of the first job. Since a global aggregate needs to be computed, a NULL-key is assigned. The reducer performs the global aggregation. The output is a single number that is written to the distributed filesystem.

- The last job combines the departments with the results of the first two jobs. As an initialization step, the result from the second job is added to an auxiliary structure called *Distributed Cache*. Values in the distributed cache are copied to each node so that they are locally accessible to each instance of the mapper. In the actual job, the mapper takes both the original XML file and the result from the first job as input. For records from the original XML file, it selects the active departments. For records from the result of the first job, it compares the employee count against the value from the distributed cache and selects only those with a larger count. It uses the same method of augmenting the key/value pairs with a lineage tag, as described in Section 3.1. The reducer then performs the join in the same fashion as the first reducer for the query given in Section 3.1.

PACT For the PACT implementation of the XQuery, we can make use of the more flexible set of operators (right-hand side of Figure 5). We first use two Map contracts to project employee and active department tuples from the XML file. The CoGroup contract groups employees by their department and compute the number of employees as well as the average salary in the department. Furthermore, an activity flag is set to true if the department is active. Subsequently a Reduce contract is used to compute the average number of employees across all departments. The resulting average number is joined with all tuples via a final Cross contract that filters for those departments which are active and have more than average employees.

3.3 K-Means Clustering Iteration

Task K-Means is a widely used data mining procedure to partition data into k groups, or clusters, of similar data points. The algorithm works by iteratively adjusting a set of k random initial cluster centers. In each iteration data points are assigned to their nearest² cluster center. The cluster centers are then recomputed as the centroid of all assigned data points. This procedure is continued until a convergence property is met.

The following pseudo code sketches the K-Means algorithm:

```

1: initialize k random initial centers
2: WHILE NOT converged
3:   FOR EACH point
4:     assign point to most similar center
5:   FOR EACH center
6:     center = centroid of assigned points
7: END

```

For the parallel implementations we will look at how to implement one iteration of K-Means. This single iteration can then be started as many times as needed from a control program.

²According to a specified distance measure, for example the Euclidean distance.

One iteration includes the following two steps:

1. Assigning each data point to its nearest center.
2. Recomputing the centers from the assigned data points.

We will assume that data is stored in two files within a distributed file system - the first containing the data points in the form of point-id/location pairs (pID/pPoint), the second one containing the cluster centers in the form of cluster-id/location pairs (cID/cPoint). For the first iteration, the cluster centers will be initialized randomly. For any subsequent one, the result file written by the previous iteration will be used. The actual point data is not of interest for us and can be of arbitrary format, as long as a distance measure and a way of computing the centroid is specified.

MapReduce The MapReduce implementation of a K-Means iteration is illustrated on the left-hand side of Figure 6. As a preparation, we need to give the mapper access to all cluster centers in order to compute the distance between the data points and the current cluster centers. Similar as in the XQuery implementation (c. f. Section 3.2), we use the distributed cache as an auxiliary structure. By adding the file containing the cluster centers to the distributed cache, the centers are effectively broadcasted to all nodes. That way the programmer “hardcodes” the join strategy between cluster centers and data points into the MapReduce program.

The Map function is invoked for each data point and computes the pairwise distance to each cluster center. As a result, it emits for each point a (cID/pPoint) pair where cID is the ID of nearest cluster center and pPoint is the location of the point. The Reduce function processes all pairs of the same cluster center and computes its new location as the centroid from all assigned points. If the distance metric permits it, a Combine function can be introduced to pre-aggregate for each center the centroid on the local machine. Since only the pre-aggregated data is then transferred between nodes, the total volume of data transfer can be significantly reduced.

PACT The PACT implementation of K-Means is depicted on the right-hand side of Figure 6. Instead of having to rely on a distributed cache, the PACTs allows us to directly express the “join” between clusters and points via the Cross contract. The user function attached to the Cross contract computes the pairwise distances between data points and clusters, emitting (pID/pPoint, CID, distance) tuples. The PACT compiler can choose between broadcasting one of the inputs or using a symmetric-fragment-and-replicate strategy to build the distributed Cartesian product of centers and data points as required for the Cross contract. Here, the common choice will be broadcasting the cluster centers, since they form a very small data set compared to the set of data points.

After the Cross function, the program uses a Reduce contract to find the minimum cluster distance for each data point and emit a (cID/pPoint) tuple for the nearest cluster. The final step is similar to the MapReduce implementation: A Reduce contract computes the new center locations, with an optional Combine function pre-aggregating the cluster centroids before the partitioning.

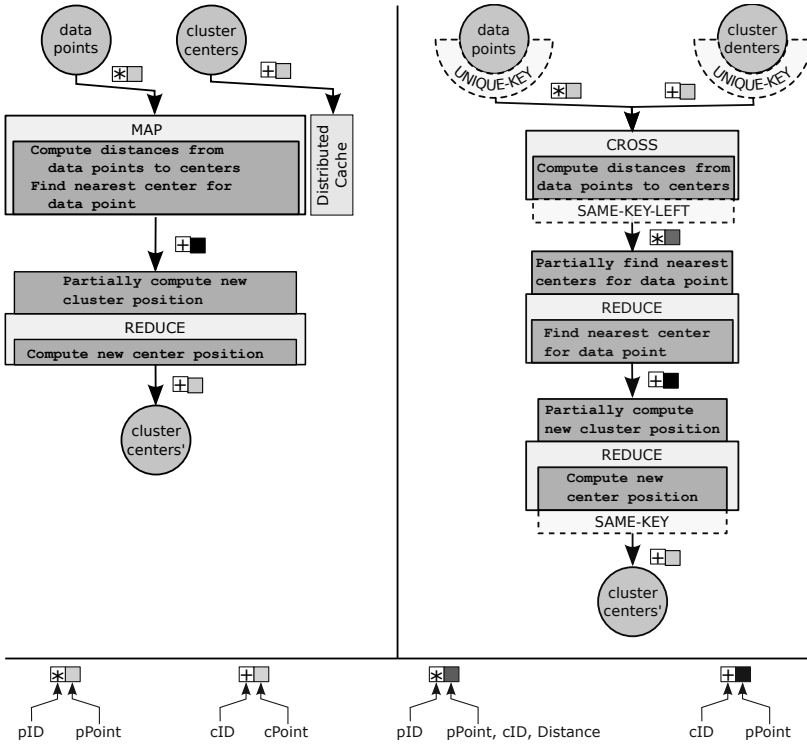


Figure 6: K-Means Iteration for MapReduce (left) and PACT (right)

The correct use of the output contracts speeds up this program significantly. Note that in Figure 6, the “data points” source is annotated with a *UniqueKey* output contract, declaring that each key/value pair produced by this source has a unique key. An implicit disjoint partitioning exists across globally unique keys. The function implementing the Cross contract declares via its output contract that it preserves the key of its left input - the data points. If the center points are broadcasted³ and the data points remain on their original node, the partitioning still exists after the cross function. Even more, all tuples for the same data point occur contiguously in the result of the Cross function. The Reduce contract needs hence neither partition nor sort the data - it is already in the correct format.

This example illustrates nicely the concept of declaring the requirements for parallelization, rather than explicitly specifying how to do it. In MapReduce, the usage of a Reduce function always implies a partitioning and sort step. For PACT, the Reduce contract merely describes that the key/value pairs need to be processed group-wise by distinct key. The compiler can infer for this program that the data is already in the required form. It directly passes the data from the output of the Cross function to the Reduce function without any intermediate processing by the system.

³This is the common strategy chosen by the optimizer

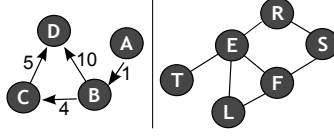


Figure 7: Example graphs for Pairwise Shortest Paths (left) and Triangle Enumeration (right)

3.4 Pairwise Shortest Paths

Task In graph theory, the pairwise shortest path problem is the determination of paths between every pair of vertices such that the sum of the lengths of its constituent edges is minimized. Consider a directed graph $G = (V, E, s, t, l)$ where V and E are the sets of vertices and edges, respectively. Associated with each edge $e \in E$ is a source vertex $v_s = s(e)$, a target vertex $v_t = t(e)$ and a length $l_e = l(e)$. The Floyd-Warshall algorithm (also commonly known as Floyd’s algorithm) is an efficient algorithm for simultaneously finding the pairwise shortest paths between vertices in such a directed and weighted graph.

```

1:   for k = 1:n
2:       for i = 1:n
3:           for j = 1:n
4:               D(i, j) = min(D(i, j), D(i, k) + D(k, j));

```

The algorithm takes the adjacency matrix D of G and compares all possible paths through the graph between each pair of vertices. It incrementally improves the candidates for the shortest path between two vertices, until the optimal is found. A parallel variant can be achieved by iteratively performing the following 4 steps until a termination criterion (number of iterations or path updates) is satisfied:

1. Generate two sets of key/value pairs:
 $P_S = \{(p.source, p) | p \in I_k\}, P_T = \{(p.target, p) | p \in I_k\}$
2. Perform an equi-join on the two sets of pairs:
 $P_J = P_T \bowtie_{end=start} P_S$
3. Union the joined set of paths with the intermediate result of the previous iteration:
 $P_U = P_J \cup I_k$
4. For all pairwise distances keep the paths with minimal length:

$$I_{k+1} = \{\min_{i,j} \{ (a, b) \in P_U | a = i \wedge b = j \} \}_{length}$$

Here I_k is the set of the shortest paths in the k -th iteration.

MapReduce The proposed MapReduce variant of Floyd-Warshall algorithm consists of a driver program which initiates iterations until a termination criterion is satisfied and two successive MapReduce jobs which alternately join the intermediate shortest paths and determine the paths of minimal length. The MapReduce jobs are shown on the left-hand

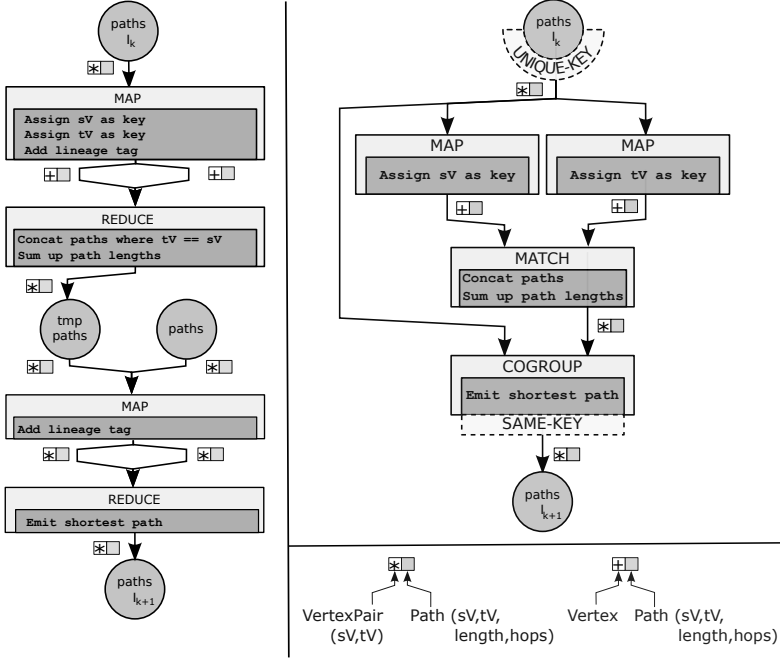


Figure 8: Implementation of the Shortest Paths Algorithm with MapReduce (left) and PACT (right)

side in Figure 8.

We assume that the input data set I_k is structured the following way: Source and target vertex of a path build a composed key, whereas the hops and length of the path from the source to the target vertex are stored in the value. For the initial iteration ($k = 0$), these paths are the edges from the original graph.

The first Map task processes the inputs set I_k and generates the sets P_S and P_T by emitting each path twice: The paths emitted into P_S use the source vertex of the path as key, the paths emitted into P_T use the target vertex of the path as key. The value is in both cases a description of the path containing the length and the hops. Since the mapper has only one output, a lineage tag is added to each path, describing the set it belongs to (c. f. Section 3.1). The subsequent reducer collects all paths sharing the same key and separates them into the subsets of P_T and P_S . It concatenates the paths of each element in P_T with each element in P_S , thereby effectively joining the paths and creating new candidates for the path that starts at the beginning of the path described by the element from P_T and ending at the end of the path described by the element from P_S . The reducer emits those paths in the same format as the input data, using the source and target vertex together as the key.

The mapper of the second job is basically an identity function. It takes the union of the original input I_k and the result of the first reducer as its inputs and re-emits it. The subsequent reducer receives all paths that start and end at the same vertices in one group. It keeps only the paths with minimal length and emits the set I_{k+1} as result of the k -th iteration.

PACT The right-hand side of Figure 8 shows the PACT variant of the same algorithm. The first mapper of the MapReduce job is substituted by two individual map tasks: the first one emits the source vertex of the path as key, the second one the target vertex. The Match performs the same task as reducer in the MapReduce variant - the join of the sets P_S and P_T , although in a more natural fashion. Finally, the CoGroup contract guarantees that all pairs (with the same key) from two different inputs are supplied to the minimum aggregation. It does so by grouping the output of the Match by individually, selecting the shortest of the new paths for each pair of vertices. Lastly, it compares the length of that path with the original shortest path's length and returning the shorter.

3.5 Edge-Triangle Enumeration

Task Identifying densely-connected subgraphs or *trusses* [Coh08] within a large graph is a common task in many use-cases such as social network analysis. A typical preprocessing step is to enumerate all triangles (3-edge cycles) in the graph. For simplicity, we consider only undirected graphs, although the algorithm can be extended to handle more complex graph structures (like multigraphs) with the help of a simplifying preprocessing step. The algorithm requires a total order over the vertices to be defined, for example a lexicographical ordering of the vertex IDs. The graph is stored in the distributed filesystem as a sequence of edges (pairs of vertices). When generating the key/value pairs from the file, each edge will be its own pair. Inside the pair, both the key and value will consist of both the edge's vertices, ordered by the defined ordering.

MapReduce The MapReduce approach to solve the edge-triangle enumeration problem was proposed by Cohen [Coh09]. It requires the graph to be represented as a list of edges, augmented with the degrees of the vertices they connect. The implementation is depicted on the left-hand side of Figure 9 and comprises two successive MapReduce jobs that enumerate and process the so-called *open triads* (pairs of connected edges) of the graph. The first Map task sets for each edge the key to its lower degree vertex. The subsequent reducer works on groups of edges sharing a common lower-degree vertex and outputs each possible subset consisting of two edges, using the vertex pair defined by the ordering of the two corresponding higher-degree vertices as the key. The mapper of the second job takes two inputs – the original augmented edge list and the open triads from the preceding reducer. It sets the edge's vertices (in order) as the key for the original edges and leaves the open triads unchanged. The technique of adding a lineage tag to each record is used, allowing the second reducer to separate its inputs again into sets of open triads and edges. Hence, it works on groups consisting of zero or more open triads and at most one single edge which completes the triads forming a closed 3-cycle.

PACT The edge-triangle enumeration algorithm can be expressed as a PACT program as shown on the right-hand side of Figure 9. The first MapReduce job, enumerating all open triads, can be reused without any further change. The second MapReduce job is replaced

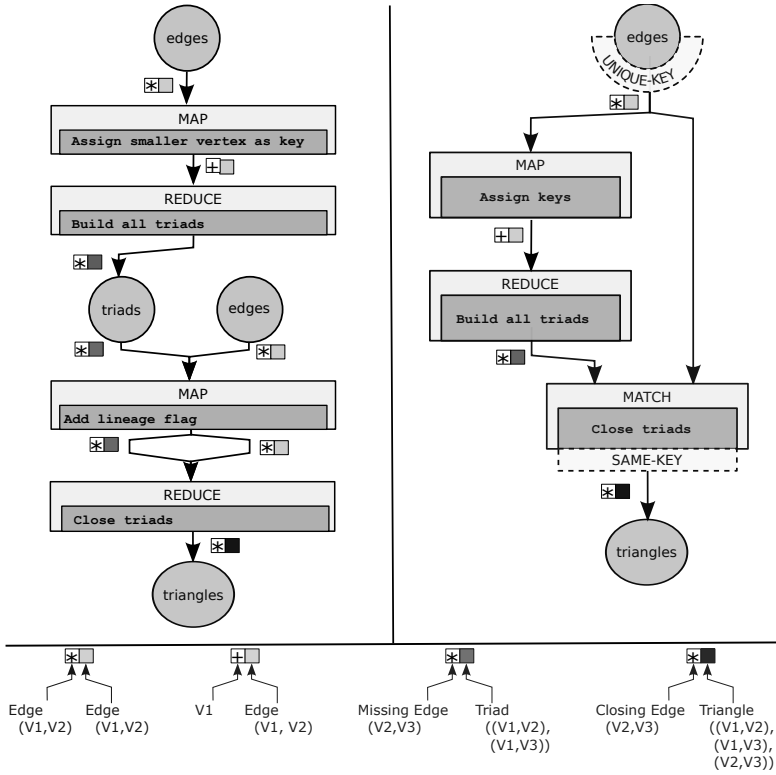


Figure 9: Enumerating triangles with MapReduce (left) and PACT (right)

by a Match contract with two inputs – the Reduce contract’s result and the original edge list. Open triads and closing edges are matched by their key. The Match function closes the triad and outputs the three edges as a closed triangle.

4 Related Work

In recent years a variety of approaches for web-scale data analysis have been proposed. All of those efforts base on large sets of shared-nothing servers and a massively-parallel job execution. However, their programming abstractions and interfaces differ significantly.

The MapReduce programming model and execution framework [DG04] are among the first approaches for data processing on the scale of several thousand machines. The idea of separating concerns about parallelization and fault tolerance from the sequential user code made the programming model popular. As a result, MapReduce and its open source implementation Hadoop [Had] have evolved as a popular parallelization platform for both industrial [TSJ⁺09, ORS⁺08, BERS] and academic research [YDHP07, VCL10].

Due to the need for ad-hoc analysis of massive data sets and the popularity of Hadoop, the research and open-source communities have developed a number of higher-level languages and programming libraries for Hadoop. Among those approaches are Hive [TSJ⁺09], Pig [ORS⁺08], JAQL [BERS], and Cascading [Cas]. Those projects have a similar goals in common, such as to ease the development of data parallel programs and to enable the reuse of code. Data processing tasks written in any of these languages are compiled into one or multiple MapReduce jobs which are executed on Hadoop. However, all approaches are geared to different use-cases and have considerably varying feature sets.

The Hive system focuses on data warehousing scenarios and is based on a data model which is strongly influenced by the relational data model. Its language for ad-hoc queries, HiveQL, borrows heavily from SQL. Hive supports a subset of the classical relational operations, such as select, project, join, aggregate, and union all. Pig's data and processing model is less tightly coupled to the relational domain. The query language Pig Latin is rather designed to fit in a sweet spot between the declarative style of SQL, and the low-level, procedural style of MapReduce [ORS⁺08]. JAQL is a query language for the JSON data model. Its syntax resembles UNIX pipes. Unlike HIVE or Pig, JAQL's primary goal is the analysis of large-scale semi-structured data.

All of the three languages apply several optimizations to a given query that the PACT compiler applies as well. The key difference is, that they deduce their degrees of freedom from the algebraic specification of their data model and its operators. In contrast, PACT's compiler deduces them from the input and output contracts, thereby maintaining schema-freeness, which is one of the main distinction between MapReduce systems and relational databases. Another important difference is that HIVE, JAQL, and Pig realize their decisions within the user code, which is nontransparent to the execution platform. The PACT compiler's decisions are known to the execution framework and hence could be adapted at runtime. All aforementioned languages could be changed to compile to PACT programs instead of MapReduce jobs, automatically benefiting from the PACT compiler optimizations. Therefore, those works are orthogonal to our.

Cascading is based on the idea of pipes and filters. It provides primitives to express for example split, join, or grouping operations as part of the programming model. Cascading's abstraction is close to the PACT programming model. However, like the declarative languages discussed above, Cascading translates its programs by directly mapping them into a sequence of MapReduce jobs. It performs only simple rewrite optimizations such as chaining map operations. Given the more flexible execution engine Nephele, the PACT compiler considers several alternative plans with different execution strategies (such as broadcasting vs. repartitioning) for a given program.

Despite the success of the MapReduce programming model, its ability to efficiently support complex data processing tasks, e.g. join-like operations, has been a major concern [YDHP07, PPR⁺09, BEH⁺10]. As a remedy, Yang et al. [YDHP07] proposed to extend the classic Map and Reduce cycle by a third, so-called Merge phase. The additional Merge phase can process data from two separate input sets and therefore enables a more natural and efficient implementation of join-like operators. Several strategies for efficient domain-specific join operations based on the unmodified version of MapReduce exist, e.g. set similarity joins [VCL10]. Other work focuses on improving Hadoop's support for iterative tasks [BHBE10].

SCOPE [CJL⁺08] and DryadLINQ [YIF⁺08] are declarative query languages and frameworks, designed to analyze large data sets. Both approaches differ from the already discussed approaches, as they do not build upon MapReduce. Instead, queries are compiled to directed acyclic graphs (DAGs) and executed by the Dryad system [IBY⁺07]. In that sense, SCOPE and DryadLINQ are similar to the PACT programming model, which is compiled to a DAG-structured Nephele schedule. However, SCOPE and DryadLINQ are higher-level languages and not offer an abstraction that enables dynamic parallelization. In contrast to MapReduce-based query languages or the Nephele/PACT system, both languages omit an explicit, generalized parallelization model. Instead, they consider parallelization on a language-specific level.

5 Conclusions

We presented a comparison of several analytical tasks in their MapReduce and PACT implementations. The PACT programming model is a generalization of MapReduce, providing additional second-order functions, and introducing output contracts. While the performance benefits of the PACT programming model were shown in previous work [BEH⁺10], this paper focused on the perspective of the programmer.

We have shown that the extended set of functions suits many typical operations very well. The following points clearly show PACT's advantages over MapReduce: 1) The PACT programming model encourages a more modular programming style. Although often more user functions need to be implemented, these have much easier functionality. Hence, interweaving of functionality which is common for MapReduce can be avoided. 2) Data analysis tasks can be expressed as straight-forward data flows. That becomes in particular obvious, if multiple inputs are required. 3) PACT frequently eradicates the need for auxiliary structures, such as the distributed cache which "brake" the parallel programming model. 4) Data organization operations such as building a Cartesian product or combining pairs with equal keys are done by the runtime system. In MapReduce such functionality must be provided by the developer of the user code. 5) Finally, PACT's contracts specify data parallelization in a declarative way which leaves several degrees of freedom to the system. These degrees of freedom are an important prerequisite for automatic optimization - both a-priori and during runtime.

References

- [ABE⁺10] Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively Parallel Data Analysis with PACTs on Nephele. *PVLDB*, 3(2):1625–1628, 2010.
- [BEH⁺10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC '10: Proceedings of the ACM Symposium on Cloud Computing 2010*, pages 119–130, New York, NY, USA, 2010. ACM.

- [BERS] K. Beyer, V. Ercegovac, J. Rao, and E. Shekita. Jaql: A JSON Query Language. URL: <http://jaql.org>.
- [BHBE10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1):285–296, 2010.
- [Cas] Cascading. URL: <http://www.cascading.org/>.
- [CJL⁺08] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2):1265–1276, 2008.
- [Coh08] Jonathan Cohen. Trusses: Cohesive Subgraphs for Social Network Analysis. <http://www2.computer.org/cms/Computer.org/dl/mags/cs/2009/04/extras/msp2009040029s1.pdf>, 2008.
- [Coh09] Jonathan Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, 11:29–41, 2009.
- [DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.
- [DQRJ⁺10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schäd. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *PVLDB*, 3(1):518–529, 2010.
- [Had] Apache Hadoop. URL: <http://hadoop.apache.org>.
- [IBY⁺07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.
- [Mah] Apache Mahout. URL: <http://lucene.apache.org/mahout/>.
- [ORS⁺08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
- [PPR⁺09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD Conference*, pages 165–178, 2009.
- [TSJ⁺09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.
- [VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 495–506, New York, NY, USA, 2010. ACM.
- [YDHP07] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Douglas Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD Conference*, pages 1029–1040, 2007.
- [YIF⁺08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, pages 1–14, 2008.

Parallel Sorted Neighborhood Blocking with MapReduce

Lars Kolb, Andreas Thor, Erhard Rahm

Department of Computer Science, University of Leipzig, Germany
{kolb, thor, rahm}@informatik.uni-leipzig.de

Abstract: Cloud infrastructures enable the efficient parallel execution of data-intensive tasks such as entity resolution on large datasets. We investigate challenges and possible solutions of using the MapReduce programming model for parallel entity resolution. In particular, we propose and evaluate two MapReduce-based implementations for Sorted Neighborhood blocking that either use multiple MapReduce jobs or apply a tailored data replication.

1 Introduction

Cloud computing has become a popular paradigm for efficiently processing data and computationally intensive application tasks [AFG⁺09]. Many cloud-based implementations utilize the MapReduce programming model for parallel processing on cloud infrastructures with up to thousands of nodes [DG08]. The broad availability of MapReduce distributions such as Hadoop makes it attractive to investigate its use for the efficient parallelization of data-intensive tasks.

Entity resolution (also known as object matching, deduplication, or record linkage) is such a data-intensive and performance-critical task that can likely benefit from cloud computing. Given one or more data sources, entity resolution is applied to determine all entities referring to the same real world object [HS95, RD00]. It is of critical importance for data quality and data integration, e.g., to find duplicate customers in enterprise databases or to match product offers for price comparison portals.

Many approaches and frameworks for entity resolution have been proposed [BS06, EIV07, KR10, KTR10b]. The standard (naive) approach to find matches in n input entities is to apply matching techniques on the Cartesian product of input entities. However, the resulting quadratic complexity of $O(n^2)$ results in intolerable execution times for large datasets [KTR10a]. So-called blocking techniques [BCC03] thus become necessary to reduce the number of entity comparisons whilst maintaining match quality. This is achieved by semantically partitioning the input data into blocks of similar records and restricting entity resolution to entities of the same block. Sorted neighborhood (SN) is one of the most popular blocking approaches [HS95]. It sorts all entities using an appropriate blocking key and only compares entities within a predefined distance window w . The SN approach thus reduces the complexity to $O(n \cdot w)$ for the actual matching.

In this study we investigate the use of MapReduce for the parallel execution of SN block-

ing and entity resolution. By combining the use of blocking and parallel processing we aim at a highly efficient entity resolution implementation for very large datasets. The proposed approaches consider specific partitioning requirements of the MapReduce model and implement a correct sliding window evaluation of entities. Our contributions can be summarized as follows:

- We demonstrate how the MapReduce model can be applied for the parallel execution of a general entity resolution workflow consisting of a blocking and matching strategy.
- We identify the major challenges and propose two approaches for realizing Sorted Neighborhood Blocking on MapReduce. The approaches (called JobSN and RepSN) either use multiple MapReduce jobs or apply a tailored data replication during data redistribution.
- We evaluate both approaches and demonstrate their efficiency in comparison to the sequential approach. The evaluation also considers the influence of the window size and data skew.

The rest of the paper is organized as follows. In the next section we introduce the MapReduce programming paradigm. Section 3 illustrates the general realization of entity resolution using MapReduce. In Section 4, we describe how the SN blocking strategy can be realized based on MapReduce. Section 5 describes the performed experiments and evaluation. Related work is discussed in Section 6 before we conclude.

2 MapReduce

MapReduce is a programming model introduced by Google in 2004 [DG04]. It supports parallel data-intensive computing in cluster environments with up to thousands of nodes. A MapReduce program relies on data partitioning and redistribution. Entities are represented by $(key, value)$ pairs. A computation is expressed with two user defined functions:

$$\begin{aligned} map &: (key_{in}, value_{in}) \rightarrow list(key_{tmp}, value_{tmp}) \\ reduce &: (key_{tmp}, list(value_{tmp})) \rightarrow list(key_{out}, value_{out}) \end{aligned}$$

These functions contain sequential code and are executed in parallel across many nodes utilizing present data parallelism. MapReduce nodes run a fixed number of mapper and/or reducer processes. Mapper processes scan disjoint input partitions in parallel and transform each entity in a $(key, value)$ -representation before the *map* function is executed. The output of a *map* function is sorted by key and repartitioned by applying a partitioning function on the key. A partition may contain different keys but all values with the same key are in the same partition. The partitions are redistributed, i.e., all $(key, value)$ pairs of a partition are sent to exactly one node. The receiving node hosts a fixed number of reducer

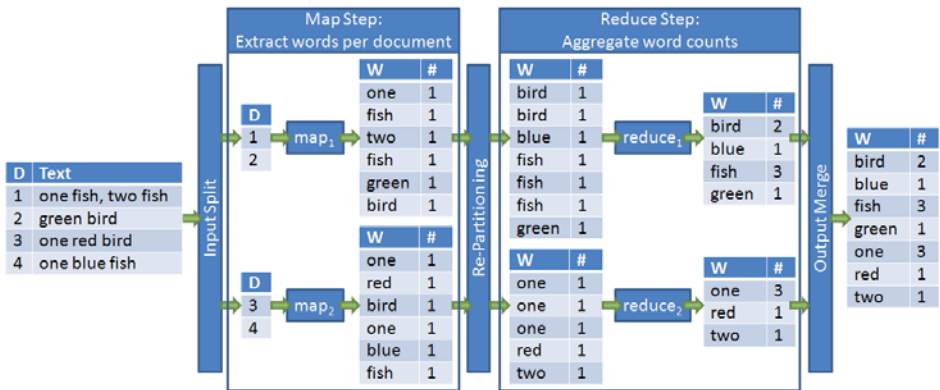


Figure 1: Example of a MapReduce program for counting word occurrences in documents (similar to [LD10])

processes whereas a single reducer is responsible for handling the *map* output pairs from all mappers that share the same key. Since the number of keys within a dataset is in general much higher than the number of reducers, a reducer merges all incoming (*key, value*) pairs in a sorted order by their intermediate keys. In the reduce phase the reducer passes all values with the same key to a *reduce* call.

An exemplary data flow of a MapReduce computation is shown in Figure 1. The MapReduce program counts the number of term occurrences across multiple documents which is a common task in information retrieval. The input data (list of documents) is partitioned and distributed to the $m = 2$ mappers. In the simple example of Figure 1, two documents are assigned to each of the two mappers. However, a mapper usually processes larger partitions in practice. Instances of the map function are applied to each partition of the input data in parallel. In our example, the map function extract all words for all documents and emits a list of (*term*, 1) pairs. The partitioning assigns every (*key, value*) pair to one reducer according to the key. In the example of Figure 1 a simple range partitioning is applied. All keys (words) starting with a letter from *a* through *m* are assigned to the first reducer; all other keys are transferred to the second reducer. The input partitions are sorted for all reducers. The user-defined reduce function then aggregates the word occurrences and outputs the number of occurrences per word. The output partitions of reduce can then easily be merged to a combined result since two partitions do not share any key.

There are several frameworks that implement the MapReduce programming model. Hadoop [Fou06] is the most popular implementation throughout the scientific community. It is free, easy to setup, and well documented. We therefore implemented and evaluated our approaches with Hadoop. Most MapReduce implementations utilize a distributed file system (DFS) such as the Hadoop distributed file system [Bor07]. The input data is initially stored partitioned, distributed, and replicated across the DFS. Partitions are redistributed across the DFS in the transition from map to reduce. The output of each reduce call is written to the DFS.

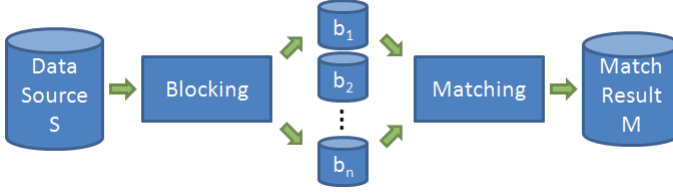


Figure 2: Simplified general entity resolution workflow

3 Entity resolution with MapReduce

In this work we consider the problem of entity resolution (deduplication) within one source. The input data source $S = \{e_i\}$ contains a finite set of entities e_i . The task is to identify all pairs of entities $M = \{(e_i, e_k) \mid e_i, e_k \in S\}$ that are regarded as duplicates.

Figure 2 shows a simplified generic entity resolution workflow. The workflow consists of a blocking strategy and a matching strategy. Blocking semantically divides a data source S into possibly overlapping partitions (blocks) b_i , with $S = \bigcup b_i$. The goal is to restrict entity comparison to pairs of entities that reside in the same block. The partitioning into blocks is usually done with the help of blocking keys based on the entities' attribute values. Blocking keys utilize the values of one or several attributes, e.g., product manufacturer (to group together all products sharing the same manufacturer) or the combination of manufacturer and product type. Often, the concatenated prefixes of a few attributes form the blocking key. A possible blocking key for publications could be the combination of the first letters of the authors' last names and the publication year (similar to the reference list in this paper).

The matching strategy identifies pairs of matching entities of the same block. Matching is usually realized by pairwise similarity computation of entities to quantify the degree of similarity. A matching strategy may also employ several matchers and combine their similarity scores. As a last step the matching strategy classifies the entity pairs as match or non-match. Common techniques include the application of similarity thresholds, the incorporation of domain-specific selection rules, or the use of training-based models. Our entity resolution model abstracts from the actual matcher implementation and only requires that the matching strategy returns the list of matching entity pairs.

The realization of the general entity resolution workflow with MapReduce is relatively straightforward by implementing blocking within the map function and by implementing matching within the reduce function. To this end, map first determines the blocking key for each entity. The MapReduce framework groups entities with the same blocking key to blocks and redistributes them. The reduce step then matches the entities within one block. Such a procedure shares similarities with the join computation in parallel database systems [DG92]. There, the join key (instead of the blocking key) is used for data repartitioning to allow a subsequent parallel join (instead of match) computation. The join (merge) results are disjoint by definition and can thus easily merged to obtain the complete result.

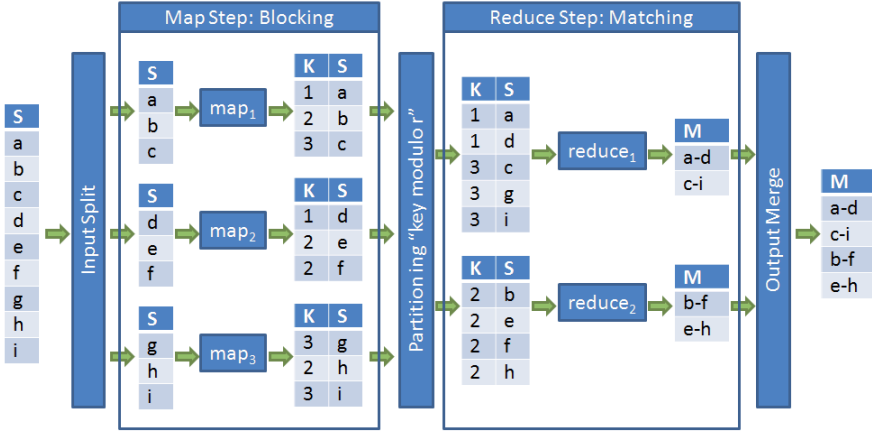


Figure 3: Example of a general entity resolution workflow with MapReduce ($n = 9$ input entities, $m = 3$ mappers, $r = 2$ reducers)

Figure 3 illustrates an example for $n=9$ entities, $a-i$, of an input data source S using $m=3$ mappers and $r=2$ reducers. First, the input partitioning (split) divides the input source S into m partitions and assigns one partition to each mapper. Then, the individual mappers read their (preferably) local data in parallel and determine a blocking key value K for each of the input entities.¹ For example, entity a has blocking key value 1. Afterwards all entities are dynamically redistributed by a partition function such that all entities with the same blocking key value are sent to the same reducer (node). In the example of Figure 3, blocking key values 1 and 3 are assigned to the first reducer whereas key 2 is assigned to the second node. The receivers group the incoming entities locally and identify the duplicates in parallel. For example, the first reducer identifies the duplicate pairs (a, d) and (c, i) . The reduce outputs can finally be merged to achieve the overall match result.

Unfortunately, the sketched MapReduce-based entity resolution workflow has several limitations:

Disjoint data partitioning: MapReduce uses a partition function that determines a single output partition for each map output pair based on its key value. This approach is suitable for many blocking techniques but complicates the realization of blocking approaches with overlapping blocks. For example, the sorted neighborhood approach does not only compare entities sharing the same blocking key.

Load balancing: Blocking may lead to partitions of largely varying size due to skewed key values. Therefore the execution time may be dominated by a single or a few reducers. Load balancing and skew handling is a well-known problem in parallel database systems [DNSS92]. The adaptation of those techniques to the MapReduce paradigm is beyond the scope of this paper and left as a subject for future work.

¹Figure 3 omits the map input keys for simplicity.

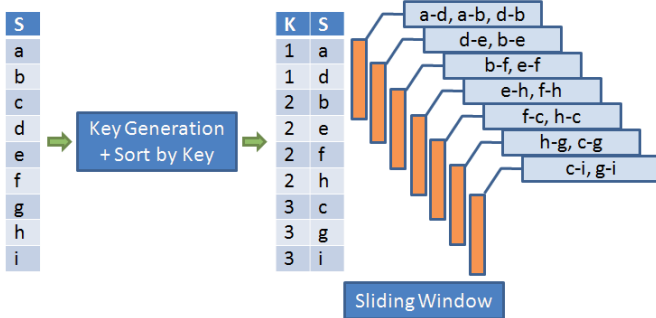


Figure 4: Example execution of sorted neighborhood with window size $w = 3$

Memory bottlenecks: All entities within the same block are passed to a single reduce call using an iterator. The reducer can only process the data row-by-row similar to a forward SQL cursor. It does not have any other options for data access. On the other hand, the matching requires that all entities within the same reduce block are compared with each other. The reducer must therefore store all entities in main memory (or must make use of other external memory) which can lead to serious memory bottlenecks. The memory bottleneck problem is partly related to the load balancing problem since skewed data may lead to large blocks which tighten the memory problem. Possible solutions have been proposed in [VCL10]. However, memory issues can also occur with a (perfect) uniform key distribution.

In this work we focus on the first challenge and propose two approaches how the popular and efficient Sorted Neighborhood blocking method SN can be realized within a MapReduce framework. As we will discuss, the SN approach is less affected by load balancing problems. Moreover, the risk for memory bottlenecks is reduced since the row-by-row process matches the SN’s sliding window approach very well.

4 Sorted Neighborhood with MapReduce

Sorted neighborhood (SN) [HS95] is a popular blocking approach that works as follows. A blocking key K is determined for each of n entities. Typically the concatenated prefixes of a few attributes form the blocking key. Afterwards the entities are sorted by this blocking key. A window of a fixed size w is then moved over the sorted records and in each step all entities within the window, i.e., entities within a distance of $w - 1$, are compared.

Figure 4 shows a SN example execution for a window size of $w = 3$. The input set consists of the same $n = 9$ entities that have already been employed in the example of Figure 3. The entities ($a-i$) are first sorted by their blocking keys (1, 2, or 3). The sliding window then starts with the first block (a, d, b) resulting in the three pairs (a, d), (a, b), and (d, b) for later comparisons. The window is then moved by one step to cover the block (d, e, b).

This leads to two additional pairs (d, e) and (b, e) . This procedure is repeated until the window has reached the final block (c, g, i) . Figure 4 lists all pairs generated by the sliding window. In general, the overall number of entity comparisons is $(n - w/2) \cdot (w - 1)$.

The SN approach is very popular for entity resolution due to several advantages. First, it reduces the complexity from $O(n^2)$ (matching n input entities without blocking) to $O(n) + O(n \cdot \log n)$ for blocking key determination and sorting and $O(n \cdot w)$ for matching. Thereby matching large datasets becomes feasible and the window size w allows for a dedicated control of the runtime. Second, the SN approach is relatively robust against a suboptimal choice of the blocking key since it is able to compare entities with a different (but similar) blocking key. The SN approach may also be repeatedly executed using different blocking keys. Such a multi-pass strategy diminishes the influence of poor blocking keys (e.g., due to dirty data) whilst still maintaining the linear complexity for the number of possible matches. Finally the linear complexity makes SN more robust against load balancing problems, e.g., if many entities share the same blocking key.

The major difference of SN in comparison to other blocking techniques is that a matcher does not necessarily only compare entities sharing the same blocking key. For example, entities d and b have different blocking keys but need to be compared according to the sorted neighborhood approach (see Figure 4). On the other hand, one of the key concepts of MapReduce is that map input partitions are processed independently. This allows for a flexible parallelization model but makes it challenging to group together entities within a distance of w since a mapper has no access to the input partition of other mappers.

Even if we assume that a mapper can determine the relevant entity sets for each entity², the general approach as presented in Section 3 is not suitable. This is due to the fact that the sliding window approach of SN leads to heavily overlapping entity sets for later comparison. In the example of Figure 4, the sliding window produces the blocks $\{a, d, b\}$ and $\{d, b, e\}$ among others. The general MapReduce-based entity resolution approach is, of course, applicable, but would expend unnecessary resources. First of all, almost all entities appear in w blocks and would therefore appear w times in the map output. Finally, the overlapping blocks would cause the generation of duplicate pairs in the reduce step, e.g., (d, b) in the above mentioned example.

We therefore target a more efficient MapReduce-based realization of SN and, thus, adapt the approach described in Section 3. The map function determines the blocking key for each input entity independently. The map output is then distributed to multiple reducers that implement the sliding window approach for each reduce partition. For example, in the case of two reducers one may want to send all entities of Figure 4 with blocking key ≤ 2 to the first reducer and the remaining entities to the second reducer. The analysis of this scenario reveals that we have to solve mainly two challenges to implement a MapReduce-based SN approach.

Sorted reduce partitions: The SN approach assumes an ordered list of all entities based on their blocking keys. A repartitioning must therefore preserve this order, i.e., the map output has to make sure that all entities assigned to reducer R_x have a smaller (or equal) blocking key than all entities of reducer R_{x+1} . This allows each reducer

²For example, this could be realized by employing a single mapper only.

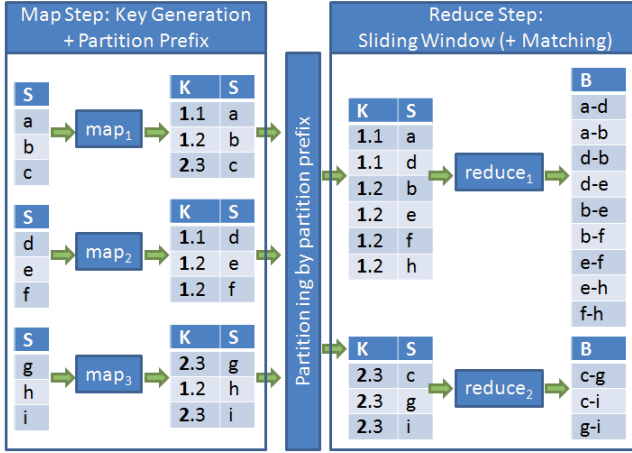


Figure 5: Example execution of sorted data partitioning with a composite key consisting of a blocking key and a partition prefix. The composite key ensures that the reduce partitions are ordered. If the sliding window approach ($w = 3$) is applied to both reduce partitions, it is only able to identify 12 out of the 15 SN correspondences (as shown in Figure 4). The pairs (f, c) , (h, c) , and (h, g) can not be found since the involved entities reside in different reduce partitions.

to apply the sliding window approach on its partition. We will address the sorted data repartitioning by employing a composite key approach that relies on a partition prefix (see Section 4.1).

Boundary entities: The continuous sliding window of SN requires that not only entities within a reduce partition but also across different reduce partitions have to be compared. More precisely, the highest $v < w$ entities of a reduce partition R_x need to be compared with the $w - v$ smallest entities of the succeeding partition R_{x+1} . In the following, we call those entities *boundary entities*. For simplicity we assume that there is no partition that holds less than w entities. Therefore it is sufficient to only compare entities of two succeeding reducers what is surely the common case. We propose two approaches (JobSN and RepSN) that employ multiple MapReduce computation steps and data replication, respectively, to process boundary entities and, thus, to map the entire SN algorithm to a MapReduce computation (Sections 4.2 and 4.3).

4.1 Sorted Reduce Partitions

We achieve sorted reduce partitions (SRP) by utilizing an appropriate user-defined function p for data redistribution among reducers in the map phase. Data redistribution is based on

the generated blocking key k , i.e., p is a function $p : k \rightarrow i$ with $1 \leq i \leq r$ and r is the number of reducers. A monotonically increasing function p (i.e., $p(k_1) \geq p(k_2)$ if $k_1 \geq k_2$) ensures that all entities assigned to reducer i have a smaller or equal blocking key than any entity processed by reducer $i + 1$.

The range of possible blocking key values is usually known beforehand for a given dataset because blocking keys are typically derived from numeric or textual attribute values. In practice simple range partitioning functions p may therefore be employed.

The execution of SRP is illustrated in Figure 5 for $m = 3$ mappers and $r = 2$ reducers. It uses the same entities and blocking keys as the example of Figure 4. In this example the function p is defined as follows: $p(k) = 1$ if $k \leq 2$, otherwise $p(k) = 2$. The map function first generates the blocking key k for each input entity and adds $p(k)$ as a prefix. In the example of Figure 5, the blocking key value for c is 3 and $p(k) = 2$. This results in a combined key value 2.3. The partitioning then distributes the $(key, value)$ pairs according to the partition prefix of the key. For example, all keys starting with 2 are assigned to the second reducer. Moreover, the input partitions for each reducer are sorted by the (combined) key. Since all keys of reducer i start with the same prefix i , the sorting of the keys is practically done based on the actual blocking key.

Afterwards the reducer can run the sliding window algorithm and, thus, generates the correspondences of interest. Figure 5 illustrates the resulting correspondences as reduce output ($B=$ Blocking). For entity resolution the reduce function will apply a matching approach to the correspondences. Reduce will therefore likely return a small subset of B . However, since we investigate in blocking techniques we leave B as output to allow for comparison with other approaches (see Section 4.2 and 4.3).

The sole use of SRP does not allow for comparing entities with a distance $\leq w$ that spread over different reducers. For example, standard SN determines the correspondence (h, c) (see Figure 4) that can not be generated since h and c are assigned to different reducers. For r reducers and a window size w , SRP misses $(r - 1) \cdot w \cdot (w - 1)/2$ boundary correspondences. We therefore present two approaches, JobSN and RepSN, that build on SRP but are also able to deal with boundary entities.

4.2 JobSN: Sorted Neighborhood with additional MapReduce job

The JobSN approach utilizes SRP and employs a second MapReduce job afterwards that completes the SN result by generating the boundary correspondences. JobSN makes thereby use of the fact that MapReduce provides sorted partitions to the reducer. A reducer can therefore easily identify the first and the last $w - 1$ entities during the sequential execution. Those entities have counterparts in neighboring partitions, i.e., the last $w - 1$ entities of a reducer relate to the first $w - 1$ entities of the succeeding reducer. In general, all reducers output the first and last $w - 1$ entities with the exception of the first and the last reducer. The first (last) reducer only returns the last (first) $w - 1$ entities.

The pseudo-code for JobSN is shown in the appendix in Algorithm 1. Figure 6 illustrates a JobSN execution example. It uses the same data of Figure 5. The map step of the first

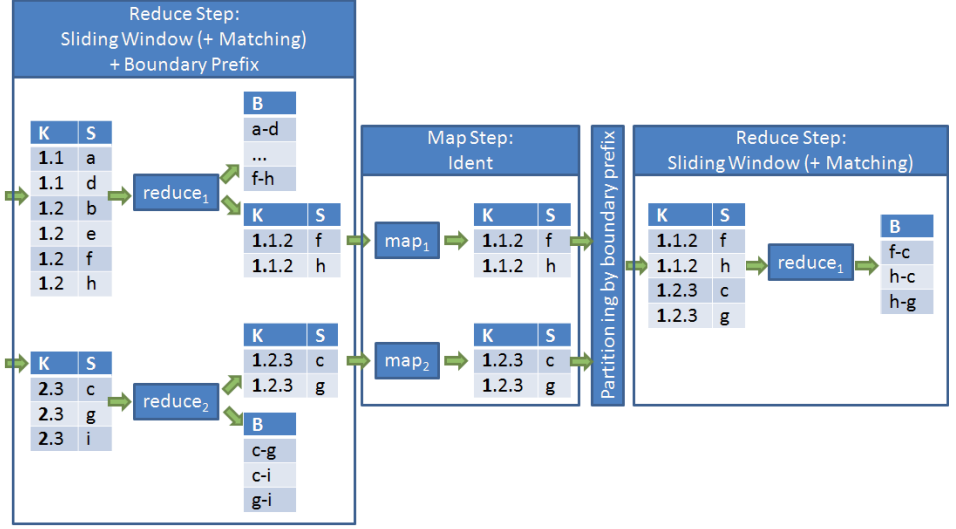


Figure 6: Example execution of SN with additional MapReduce job (JobSN, $w = 3$). The far left box is the reduce step of the first job. Its output is the input to the second MapReduce job.

job is identical with SRP of Figure 5 and omitted in Figure 6 to save space. The reduce step is extended by an additional output. Besides the list of blocking correspondences B , the reducer also emits the first and last $w - 1$ entities.

JobSN realizes the assignment of related boundary elements with an additional boundary prefix that specifies the boundary number. Since the last $w - 1$ entities of reducer $i < r$ refer to the i^{th} boundary, the keys of the last $w - 1$ entities are prefixed with i . On the other hand, the first $w - 1$ entities of the succeeding reducer $i + 1$ also relate to the i^{th} boundary. Therefore the keys of the first $w - 1$ entities of reducer $i > 1$ are prefixed with $i - 1$. The first reducer in the example of Figure 6 prefixes the last entities (f and h) with 1 and the second reducer prefixes the first entities (c and g) with 1, too. Thereby the key reflects data lineage: The actual blocking key of entity c is 3 (see, e.g., Figure 4), it was assigned to reducer number 2 during the SRP (Figure 5), and it is associated with boundary number 1 (Figure 6).

The second MapReduce job of JobSN is straightforward. The map function leaves the input data unchanged. The map output is then redistributed to the reducers based on the boundary prefix. The reduce function then applies the sliding window but filters correspondences that have already been determined in the first MapReduce job. For example, (f, h) does not appear in the output of the second job since this pair is already determined by SRP. As mentioned above, this knowledge is encoded in the lineage information of the key because those entities share the same partition number.

The JobSN approach generates the complete SN result at the expense of an additional

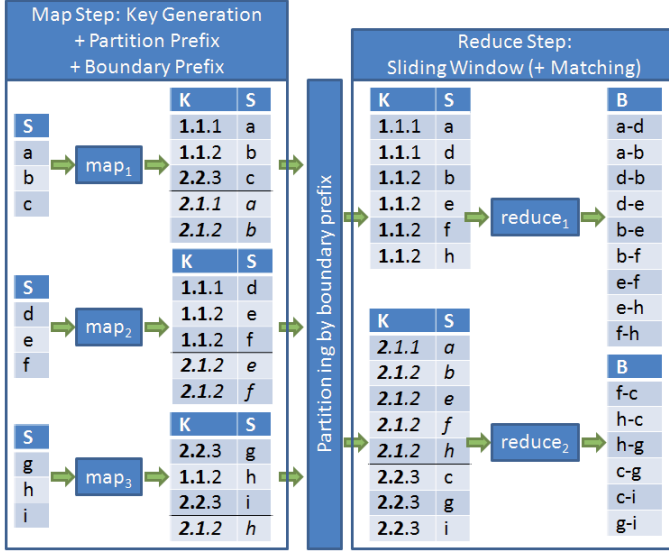


Figure 7: Example execution of sorted neighborhood with entity replication (RepSN, $w = 3$). Entities are replicated within in the map function (below solid line). Replicated entities are written in *italic type*.

MapReduce job. We expect the overhead for an additional job to be acceptable and we will evaluate JobSN’s performance in Section 5.

4.3 RepSN: Sorted Neighborhood with entity replication

The RepSN approach aims to realize SN within a single MapReduce job. It extends SRP by the idea that each reducer $i > 1$ needs to have the last $w - 1$ entities of the preceding reducer $i - 1$ in front of its input. This would ensure that the boundary correspondences appear in the reducer’s output. However, the MapReduce paradigm is not designed for mutual data access between different reducers. MapReduce only provides options for controlled data replication within the map function.

The RepSN approach therefore extends the original SRP map function so that map replicates an entity that should be send to both the respective reducer and its successor. For all but the last reduce partition r , the map function thus identifies the $w - 1$ entities with the highest blocking key k . It first outputs all entities and adds the identified boundary entities afterwards. Similar to SRP, an entity key is determined by the blocking key k plus a partition prefix $p(k)$. To distinguish between original entities and replicated boundary entities, RepSN adds an additional boundary prefix. For all original entities this boundary prefix is the same as the partition number, i.e., the composite key is $p(k).p(k).k$. The

boundary prefix for replicated entities is the partition number of the succeeding reducer, i.e., the composite key is $(p(k) + 1).p(k).k$.

RepSN is described in the appendix in Algorithm 2. Figure 7 illustrates an example execution of RepSN. The example employs $r = 2$ reducers and window size $w = 3$. Therefore all mappers identify the $w - 1 = 2$ entities with the highest key of partition 1. The output of each map function is divided into two parts. The upper part (above the solid line) is equivalent to the regular map output of SRP. The only (technical) difference is that the partition prefix is duplicated. The lower part (below the solid line) of the map output contains the replicated entities. Consider the second map function: All three entities (d , e , and f) are assigned to the partition 1 and e and f are replicated because they have the highest keys. The keys of the replicated data start with the succeeding partition 2. This ensures that e and f are sent to both reducer 1 and reducer 2.

The map output is then redistributed to the reduce functions based on the boundary prefix. Furthermore, MapReduce provides a sorted list as input to the reduce functions. Due to the structure of the composite key, the replicated entities appear at the beginning of each reducer input. Replicated entities share the same boundary prefix but have a smaller partition prefix. The reduce function then applies the sliding window approach but only returns correspondences involving at least one entity of the actual partition.

In the example of Figure 7, input and output of the first reducer are equivalent to SRP (see Figure 5). The second reducer receives a larger input partition. It ignores all replicated entities but the $w - 1 = 2$ highest (f and h). The output is the union of the corresponding SRP output and the corresponding boundary reduce output of JobSN.

RepSN allows for an entire sorted neighborhood computation within a single MapReduce job at the expense of some data replication. Since the MapReduce model does not provide any global data access³ during the computation, it is not possible to identify only the necessary entities for processing the boundary elements. Rather each map function has to identify and replicate possibly relevant entities based on its local data. Each mapper has to replicate $w - 1$ entities for all but the last partition. The maximum number of replicated entities is therefore $m \cdot (r - 1) \cdot (w - 1)$. This number is independent from the size n of input entities and may therefore be comparatively small for large datasets. We will evaluate the overhead of data replication and data transfer in Section 5. In particular we will compare it against the JobSN overhead for scheduling and executing an additional MapReduce job.

5 Experiments

We conducted a set of experiments to evaluate the efficiency of the proposed approaches. After a description of the experimental setup we study the scalability of our Sorted Neighborhood approaches. Afterwards we will discuss the effects of data skew and show its influence on execution time.

³Hadoop as the most popular implementation MapReduce offers a so called distributed cache. However, the primary purpose of this mechanism is to upfront copy read-only data (like files or archives) needed by the job to the particular nodes.

5.1 Experimental setup

We run our experiments on up to four nodes with two cores. Each node has an Intel(R) Core(TM)2 Duo E6750 2x2.66GHz CPU, 4GB memory and runs a 64-bit Debian GNU/Linux OS with a Java 1.6 64-bit server JVM. On each node we run Hadoop 0.20.2. Following [VCL10] we made the following changes to the Hadoop default configuration: We set the block size of the DFS to 128MB, allocated 1GB to each Hadoop daemon and 1GB virtual memory to each map and reduce task. Each node was configured to run at most two map and reduce tasks in parallel. Speculative execution was turned off. Both master daemons for managing the MapReduce jobs and the DFS run on a dedicated server. We used Hadoop’s SequenceFileOutputFormat with native bzip2 block compression to serialize the output of mappers and reducers that was further processed. Sequence files can hold binary $(key, value)$ pairs what conceptually allowed us to deal with $(String, String[])$ instead of $(String, String)$ pairs. Hence, we could directly access the i^{th} attribute value of an entity during matching in comparison to split a string at runtime.

The input dataset⁴ for our experiments contains about 1.4 Mio. publication records. To compare two publications we executed two matchers (edit distance on title, TriGram on abstract) and calculated the average of the two results. Pairs of entities with an average similarity score of at least 0.75 were regarded as matches. We applied an internal optimization by skipping the execution of the second matcher if the similarity after the execution of the first matcher was too low (i.e., ≤ 0.5) for reaching the combined similarity threshold. To group similar entities into blocks we used the lowercased first two letters of the title as blocking key.

5.2 Sorted Neighborhood

We evaluate the absolute runtime and the relative speedup using two window sizes of 10 and 1000. The additional MapReduce job of JobSN was executed with one reducer ($r = 1$). To ensure comparability for different numbers of mappers and reducers we used the same manually defined function in each experiment. It partitions the set of entities into 10 blocks and targets to assign the same number of entities to each block. The resulting 10 reduce tasks are executed by at most 8 reducers (see Section 5.3 for further discussion).

Figure 8 shows execution times and speedup results for up to 8 mappers and 8 reducers for the two proposed implementations. The configurations with $m = r = 1$ refers to sequential execution on a single node, the one with $m = r = 2$ refers to the execution on a single node utilizing both cores and so on. For the small window size $w = 10$, RepSN slightly outperforms JobSN due to the scheduling overhead of JobSN for the additional MapReduce job. The execution time of both RepSN and LocSN could be reduced from approximately 10.5 to about 2.5-3 minutes resulting in a relative speedup of up to 4 for 8 cores. For the larger window size $w = 1000$, the execution times scale almost linearly, for instance the execution time for RepSN could be reduced from approximately 9 to merely

⁴<http://asterix.ics.uci.edu/data/csx.raw.txt.gz>

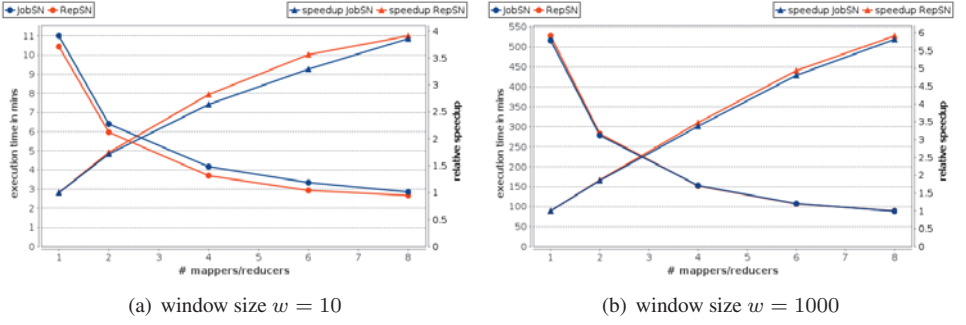


Figure 8: Comparison of the two Sorted Neighborhood implementations

1.5 hours. We observe a nearly linear speedup for the entire range of up to 4 nodes and 8 cores. The runtime of the different implementations differ only slightly. Differences can only be observed for a small amount of parallelism, i.e., RepSN was 10 minutes slower for $w = 1000$ in the sequential case. Beginning with $m = r = 4$ RepSN completed faster than JobSN. The reasons for the suboptimal speedup values (about 6 for 8 cores) are caused by design and implementation choices of MapReduce/Hadoop to achieve fault tolerance, e.g., materialization of (intermediate) results between map and reduce.

5.3 Data skew

We finally study the effects of data skew and use RepSN for this experiment. Practical data skew handling has been studied in the context of parallel DBMS [DNSS92] but has not yet been incorporated in our implementation. Similar to the hash join computation in parallel DBMS, the application of any partitioning (hash) function p as described in Section 4.1 is susceptible to data skew and resulting load imbalances. This is because the combination of blocking key skew together with key-based partitioning may lead to partitions of largely varying size so that the total execution time is dominated by a single or few reduce task. One can, of course, reduce the impact of the key skew by choosing a good partitioning function that assigns a different number of keys to the individual reduce tasks and, thus, tries to balance the number of entities per reduce task.⁵ However, the impact of a partitioning function is limited due to the following two restrictions. First, an arbitrary assignment of blocking keys to reduce tasks is not possible because SN requires sorted reduce partitions. Second, processing (very) large blocks can not be distributed to multiple reduce tasks because the MapReduce paradigm requires entities sharing the same blocking key to be processed within the same reduce task.

We ran our experiments on all 4 nodes (8 mappers and 8 reducers) with a window size

⁵Hadoop comes with features (InputSampler and TotalOrderPartitioner) that allows to sample the output of a MapReduce job and to estimate a suitable partitioning function p that avoids varying sized map output partitions and ensures totally ordered keys.

p	g
Manual	0.13
Even10	0.30
Even8	0.32
Even8.40	0.42
Even8.55	0.54
Even8.70	0.63
Even8.85	0.76

Table 1: Partitioning functions and resulting data skew

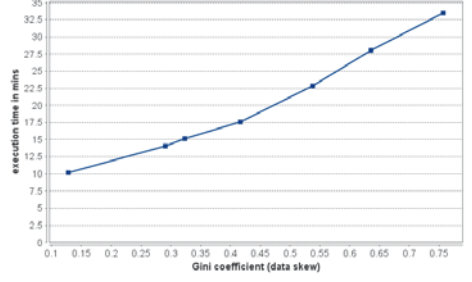


Figure 9: Execution time of RepSN for various degrees of data skew ($w = 100$)

Figure 10: Influence of data skew ($m = r = 8$)

$w = 100$. To quantify the inequality of the key distribution in the dataset we utilize the Gini coefficient $g = \frac{2 \cdot \sum_{i=1}^n i \cdot y_i}{n \cdot \sum_{i=1}^n y_i} - \frac{n+1}{n}$, whereas y_i is the number of entities in partition i and $y_i \leq y_{i+1}$. A value of 0 expresses total equality and a value of 1 maximal inequality. We evaluated the partition strategies shown in Table 1 exhibiting different degrees of data skew as indicated by their Gini coefficient. In addition to the manually defined partitioning function used in Section 5.2 with almost equally-sized partitions we evenly partitioned the key space into 10 and 8 intervals (Even10, Even8). Finally we used Even8 but modified the blocking keys so that 40%, 55%, 70% and 85%, respectively, of all entities fall in the last partition. The runtime results for the different partitioning strategies are illustrated in Figure 9. The manual partitioning strategy that was tuned for equally-sized partitions performed best, while the most skewed configuration suffered from a more than threefold execution time. Even10 completed slightly (one minute) faster than Even8 because of its smaller partitions allowing the 8 reducers processing several small partitions while a large partition is matched (improved load balancing potential). For Even8.40 - Even8.85 we observe significant increases of the execution time with a rising degree of data skew. Clearly the influence of data skew will increase for larger window sizes since more entities within a partition have to be compared by one reducer.

The observed problems are MapReduce-inherent because the programming model demands that all values with the same key are processed by the same reducer. A majority of values for one or a small subset of the dataset's keys does not allow effective parallel data processing. There is no skew handling mechanism in MapReduce except the redundant execution of outstanding map or reduce task at the end of a job (speculative execution). However, this helps only to deal with partially working or misconfigured stragglers. Therefore it becomes necessary to investigate in load balancing mechanisms for the MapReduce paradigm.

6 Related work

Entity resolution is a very active research topic and many approaches have been proposed and evaluated as described in recent surveys [EIV07, KR10]. Surprisingly, there are only a few approaches that consider parallel entity resolution. First ideas for parallel matching were described in the Febrl system [CCH04]. The authors show how the match computation can be parallelized among available cores on a single node. Parallel evaluation of the Cartesian product of two sources considering the three input cases (clean-clean, clean-dirty, dirty-dirty) is described in [KL07].

[KKH⁺10] proposes a generic model for parallel processing of complex match strategies that may contain several matchers. The parallel processing is based on general partitioning strategies that take memory and load balancing requirements into account. Compared to this work [KKH⁺10] allows the execution of a match workflow on the Cartesian product of input entities. This is done by partitioning the set of input entities and generating match tasks for each pair of partitions. A match task is then assigned to any idle node in a distributed match infrastructure with a central master node. The advantage of this approach is the high flexibility for scheduling match tasks and thus for dynamic load balancing. The disadvantage is that only the matching itself is executed in parallel. Blocking is done upfront on the master node. Furthermore in this work we rely on an widely used parallel processing framework that hides the details of parallelism and therefore is less error-prone.

We are only aware of one previous approach for parallel entity resolution on a cloud infrastructure [VCL10]. The authors do not investigate Sorted Neighborhood blocking but show how a single token-based string similarity function can be realized with MapReduce. The approach is based on a complex workflow consisting of several MapReduce jobs. This approach suffers from similar load balancing problems as observed in Section 5.3 because all entities that share a frequent token are compared by one reducer⁶. In contrast to our Sorted Neighborhood approach large partitions for frequent tokens that do not fit into memory must be handled separately. This is because all entities that contain a specific token have to be compared with each other instead of comparing only entities with a maximum distance of less than w . Compared to [VCL10], we are not limited to a specific similarity function but can apply a complex match strategy for each pair of entities within a window. Furthermore as explained in Section 3 Sorted Neighborhood can be substituted with other blocking techniques, e.g., Standard Blocking or N-gram indexing.

7 Conclusions and outlook

We have shown how entity resolution workflows with a blocking strategy and a match strategy can be realized with MapReduce. We focused on parallelizing Sorted neighborhood blocking and proposed two MapReduce-based implementations. The evaluation of our approaches demonstrated their efficiency and scalability in comparison to sequential

⁶The authors could slightly reduce the data skew by redistributing data based on the infrequent prefix tokens of a record's attribute value.

entity resolution. We also pointed out the need for incorporating load balancing and skew handling mechanisms with MapReduce.

There are further limitations of MapReduce and the utilized implementation Hadoop such as insufficient support for pipelining intermediate data between map and reduce jobs. There are other parallel data processing frameworks like [WK09] that support different types of communication channels (file, TCP, in-memory) and provide a better support for different input sets. Furthermore there are concepts like [YDHP07] that propose to adapt and extend MapReduce to simplify set operations (Cartesian product) on heterogeneous datasets.

In future work we plan to investigate load balancing and data partitioning mechanisms for MapReduce.

References

- [AFG⁺09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, EECS Department, University of California, Berkeley, 2009.
- [BCC03] Rohan Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD*, volume 3, pages 25–27, 2003.
- [Bor07] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 2007.
- [BS06] Carlo Batini and Monica Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Data-Centric Systems and Applications. Springer, 2006.
- [CCH04] Peter Christen, Tim Churches, and Markus Hegland. Febrl - A Parallel Open Source Data Linkage System. In *PAKDD*, pages 638–647, 2004.
- [DG92] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
- [DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.
- [DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
- [DNSS92] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *VLDB*, pages 27–40, 1992.
- [EIV07] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate Record Detection: A Survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.
- [Fou06] Apache Software Foundation. Hadoop. <http://hadoop.apache.org/mapreduce/>, 2006.
- [HS95] Mauricio A. Hernández and Salvatore J. Stolfo. The Merge/Purge Problem for Large Databases. In *SIGMOD Conference*, pages 127–138, 1995.

- [KKH⁺10] Toralf Kirsten, Lars Kolb, Michael Hartung, Anika Gross, Hanna Köpcke, and Erhard Rahm. Data Partitioning for Parallel Entity Matching. In *8th International Workshop on Quality in Databases*, 2010.
- [KL07] Hung-Sik Kim and Dongwon Lee. Parallel linkage. In *CIKM*, pages 283–292, 2007.
- [KR10] Hanna Köpcke and Erhard Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2):197–210, 2010.
- [KTR10a] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. In *VLDB*, 2010.
- [KTR10b] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Learning-Based Approaches for Matching Web Data Entities. *IEEE Internet Computing*, 14:23–31, 2010.
- [LD10] Jimmy Lin and Chris Dyer. Data-Intensive Text Processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.
- [RD00] Erhard Rahm and Hong Hai Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.
- [VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD Conference*, pages 495–506, 2010.
- [WK09] Daniel Warneke and Odej Kao. Nephele: efficient parallel data processing in the cloud. In *SC-MTAGS*, 2009.
- [YDHP07] Hung-Chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Douglas Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD Conference*, pages 1029–1040, 2007.

A Algorithms

Algorithm 1 and Algorithm 2 show the pseudo-code for the two proposed Sorted Neighborhood implementation JobSN and RepSN introduced in sections 4.2 and 4.3. For simplicity, we use a function `StandardSN` that implements the standard Sorted Neighborhood approach, i.e., that moves the window of size w over a sorted list of entities and outputs matching entity pairs (Algorithm 1 line 9, 26 and Algorithm 2 line 31).

Throughout the two algorithms r denotes the configured number of reducers for the MapReduce job. The partitioning function $p : k \rightarrow i$ with $1 \leq i \leq r$ determines the reducer r_i to which an entity with the blocking key value k is repartitioned. A key of the form $x.y$ denotes a composed key of x and y . Composed keys are compared component-wise. The comments indicate which parts of the composite keys are used for map-side repartitioning and reduce-side grouping of entities.

For simplicity, the pseudo-code of Algorithm 1 does not filter correspondences that have been already determined in the first phase. This can be easily achieved by comparing only entities whose second component of the composed key differ. In Algorithm 2 we use two extra functions in addition to map and reduce. The function `map_configure` is executed before a mapper executes a map task and `map_close` before termination of a map task, respectively.

Algorithm 1: JobSN

```
1 // --- Phase 1 ---
2 map ( $key_{in}=unused, value_{in}=entity$ )
3   |  $k \leftarrow$  generate blocking key for  $entity$ ;
4   |  $r_i \leftarrow p(k)$ ; // reducer to which entity is assigned by p
5   | // Use composite key to partition by  $r_i$ 
6   | output ( $key_{tmp}=r_i.k, value_{tmp}=entity$ )

7 // group by  $r_i$ , order by composed key
8 reduce ( $key_{tmp}=r_i.k, list(value_{tmp})=list(entity)$ )
9   | StandardSN ( $list(entity), w$ );
10  | first  $\leftarrow$  first  $w - 1$  entities of  $list(entity)$ ;
11  | last  $\leftarrow$  last  $w - 1$  entities of  $list(entity)$ ;
12  | if  $r_i > l$  then
13  |   | bound  $\leftarrow r_i - 1$ ;
14  |   | foreach  $entity \in$  first do
15  |   |   | output ( $key_{out}=bound.r_i.k, value_{out}=entity$ )
16  | if  $r_i < r$  then
17  |   | bound  $\leftarrow r_i$ ;
18  |   | foreach  $entity \in$  last do
19  |   |   | output ( $key_{out}=bound.r_i.k, value_{out}=entity$ )

20 // --- Phase 2 ---
21 map ( $key_{in}=bound.r_i.k, value_{in}=entity$ )
22   | // Use composite key to partition by bound
23   | output ( $key_{tmp}=bound.r_i.k, value_{tmp}=entity$ )

24 // group by bound, order by composed key
25 reduce ( $key_{tmp}=bound.r_i.k, list(value_{tmp})=list(entity)$ )
26   | StandardSN ( $list(entity), w$ );
```

Algorithm 2: RepSN

```
1 map_configure
2   // list of the entities with the w-1 highest
3   // blocking keys for each partition i<r
4   foreach  $i \in \{1, \dots, r-1\}$  do
5      $rep_i \leftarrow []$ ;

6 map ( $key_{in}=unused, value_{in}=entity$ )
7    $k \leftarrow$  generate blocking key for  $entity$ ;
8    $r_i \leftarrow p(k)$ ; // reducer to which entity is assigned by p
9    $bound \leftarrow r_i$ ;
10  if  $r_i < r$  then
11    if  $sizeOf(rep_{r_i}) < w-1$  then
12      append( $rep_{r_i}, entity$ );
13    else
14       $min \leftarrow$  determine entity from  $rep_{r_i}$  with smallest blocking key;
15       $k_{min} \leftarrow$  blocking key of  $min$ ;
16      if  $k > k_{min}$  then
17        replace( $rep_{r_i}, min, entity$ );

18  // Use composite key to partition by bound
19  output ( $key_{tmp}=bound.r_i.k, value_{tmp}=entity$ )

20 map_close
21  foreach  $i \in \{1, \dots, r-1\}$  do
22     $r_i \leftarrow i$ ;
23     $bound \leftarrow r_i + 1$ ;
24    foreach  $entity \in rep_i$  do
25      // prefix key with  $r_i+1$  to assign replicated
26      // entities to succeeding reducer
27      output ( $key_{tmp}=bound.r_i.k, value_{tmp}=entity$ )

28 // group by bound, order by composed key
29 reduce ( $key_{tmp}=bound.r_i.k, list(value_{tmp}=list(entity))$ )
30   remove all entities with  $bound \neq r_i$  from the head of  $list(entity)$  except the last  $w-1$ ;
31   StandardSN ( $list(entity), w$ );
```

PigSPARQL: Übersetzung von SPARQL nach Pig Latin

Alexander Schätzle, Martin Przyjacieli-Zablocki, Thomas Hornung, Georg Lausen

Lehrstuhl für Datenbanken und Informationssysteme
Albert-Ludwigs-Universität Freiburg
Georges-Köhler Allee, Geb. 51
79110 Freiburg im Breisgau
{schaetzl, zablocki, hornung, lausen} @ informatik.uni-freiburg.de

Abstract: Dieser Beitrag untersucht die effiziente Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen. Zum Einsatz kommt hierfür das Apache Hadoop Framework, eine bekannte Open-Source Implementierung von Google's MapReduce, das massiv parallelisierte Berechnungen auf einem verteilten System ermöglicht. Zur Auswertung von SPARQL-Anfragen mit Hadoop wird in diesem Beitrag PigSPARQL, eine Übersetzung von SPARQL nach Pig Latin, vorgestellt. Pig Latin ist eine von Yahoo! Research entworfene Sprache zur verteilten Analyse von großen Datensätzen. Pig, die Implementierung von Pig Latin für Hadoop, übersetzt ein Pig Latin-Programm in eine Folge von MapReduce-Jobs, die anschließend auf einem Hadoop-Cluster ausgeführt werden. Die Evaluation von PigSPARQL anhand eines SPARQL spezifischen Benchmarks zeigt, dass der gewählte Ansatz eine effiziente Auswertung von SPARQL-Anfragen mit Hadoop ermöglicht.

1 Einleitung

Die Menge der Daten im Internet und damit auch das potentiell zur Verfügung stehende Wissen nimmt schnell zu. Leider können große Teile dieses Wissens nicht automatisiert erfasst und verarbeitet werden, da sich die Aufbereitung und Darstellung an einem menschlichen Betrachter orientiert. Das Ziel des *Semantic Web* [BHL01] ist die Erschließung und automatisierte Verarbeitung dieses Wissens. Zu diesem Zweck wurde das Resource Description Framework (RDF) [MMM04] entwickelt, ein Standard zur Repräsentation von Daten in einem maschinenlesbaren Format. SPARQL [PS08] ist die vom W3C¹ empfohlene Anfrage-Sprache für RDF.

Die zunehmende Größe von Datensätzen erfordert die Entwicklung neuer Konzepte zur Datenverarbeitung und Datenanalyse. Google entwickelte hierfür 2004 das sogenannte MapReduce-Modell [DG04], das es dem Anwender erlaubt, parallele Berechnungen auf sehr großen Datensätzen verteilt auf einem Computer-Cluster durchzuführen, ohne sich um die Details und die damit verbundenen Probleme eines verteilten Systems Gedanken machen zu müssen. Die bekannteste frei verfügbare Implementierung des MapReduce-

¹ World Wide Web Consortium – siehe [<http://www.w3.org/>]

Modells ist das Hadoop Framework², das maßgeblich von Yahoo! weiterentwickelt wird. Da die Entwicklung auf MapReduce-Ebene trotz aller Vorzüge dennoch recht technisch und anspruchsvoll ist, entwickelten Mitarbeiter von Yahoo! eine Sprache zur Analyse von großen Datensätzen mit Hadoop, Pig Latin [OI08], die dem Anwender eine einfache Abstraktionsebene zur Verfügung stellen soll. Pig, die Implementierung von Pig Latin für Hadoop, ist mittlerweile ein offizielles Subprojekt von Hadoop.

Da insbesondere auch die Menge der verfügbaren RDF-Datensätze stetig zunimmt³, müssen auch hier neue Konzepte zur Auswertung solcher Datensätze entwickelt werden. Dabei sind klassische, auf nur einem Computer ausgeführte, Systeme aufgrund der begrenzten Ressourcen zunehmend überfordert. Die grundlegende Idee dieser Arbeit ist es daher, die Mächtigkeit von Hadoop zur Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen zu nutzen. Hierfür wurde eine Übersetzung von SPARQL nach Pig Latin entwickelt und implementiert, die eine SPARQL-Anfrage in ein äquivalentes Pig Latin-Programm überführt. Dieser Ansatz hat den Vorteil, dass die Übersetzung direkt von bestehenden Optimierungen bzw. zukünftigen Weiterentwicklungen von Pig profitiert. Die wesentlichen Beiträge unserer Arbeit sind wie folgt: Zunächst definieren wir für ein ausdrucks mächtiges Fragment von SPARQL eine Übersetzung in ein äquivalentes Pig Latin-Programm. Das betrachtete Fragment deckt dabei insbesondere einen Großteil der Anfragen ab, die in der offiziellen Dokumentation [PS08] enthalten sind. Nach unserem Kenntnisstand ist diese Arbeit die erste umfassende Darstellung einer Übersetzung von SPARQL nach Pig Latin. Darüber hinaus untersuchen wir Optimierungsstrategien für die Übersetzung und bestätigen deren Wirksamkeit. Abschließend zeigen wir anhand eines SPARQL spezifischen Performance Benchmarks, dass die von uns entwickelte Übersetzung eine effiziente Auswertung von SPARQL-Anfragen auf sehr großen RDF-Datensätzen ermöglicht.

Der weitere Verlauf dieser Arbeit ist wie folgt strukturiert. In Kapitel 2 werden die nötigen Grundlagen von RDF, SPARQL, MapReduce und Pig Latin kurz dargestellt. Kapitel 3 erläutert die von uns entwickelte Übersetzung von SPARQL nach Pig Latin. In Kapitel 4 folgt die Evaluation der Übersetzung mit einem SPARQL spezifischen Performance Benchmark und Kapitel 5 gibt einen Überblick über verwandte Arbeiten. Abschließend werden die Ergebnisse der Arbeit in Kapitel 6 zusammengefasst.

2 Grundlagen

Dieses Kapitel stellt die Grundlagen der von uns verwendeten Technologien kurz dar.

2.1 RDF

Das Resource Description Framework (RDF) ist ein vom World Wide Web Consortium (W3C) entwickelter Standard zur Modellierung von Metainformationen über beliebige Ressourcen (z.B. Personen oder Dokumente). Eine ausführliche Darstellung von RDF

² siehe [<http://hadoop.apache.org>]

³ siehe [<http://esw.w3.org/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics>]

findet sich in [MMM04]. Im Folgenden werden nur die grundlegenden Konzepte von RDF kurz vorgestellt.

RDF-Tripel. Grundlage der Wissensrepräsentation in RDF sind Ausdrücke der Form <Subjekt, Prädikat, Objekt>. Ein RDF-Tripel lässt sich folgendermaßen interpretieren:

<Subjekt> hat die Eigenschaft <Prädikat> mit dem Wert <Objekt>.

RDF-Tripel können URIs, Blank Nodes und RDF-Literale enthalten. URIs (Uniform Resource Identifier) sind weltweit eindeutige Bezeichner für Ressourcen (z.B. <http://example.org/Peter>), Blank Nodes sind lokal eindeutige Bezeichner (z.B. _:address) und RDF-Literale sind atomare Werte (z.B. "27").

RDF-Graph. Ein RDF-Dokument besteht im Wesentlichen aus einer Abfolge von RDF-Tripeln und lässt sich als gerichteter Graph interpretieren. Jedes Tripel entspricht dabei einer beschrifteten Kante (Prädikat) von einem Knoten im Graph (Subjekt) zu einem anderen Knoten (Objekt).

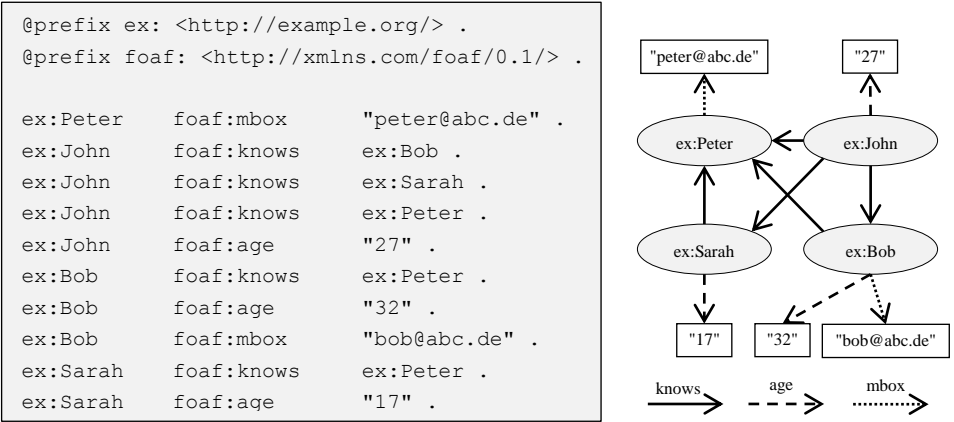


Abbildung 1: RDF-Dokument mit entsprechendem RDF-Graph

2.2 SPARQL

SPARQL⁴ ist die vom W3C empfohlene Anfragesprache für RDF. Die folgende kurze Darstellung beruht auf der offiziellen Dokumentation von SPARQL aus [PS08]. Eine formale Definition der Semantik von SPARQL findet sich ebenfalls in der offiziellen Dokumentation oder in [PAG09].

Graph Pattern. Eine SPARQL-Anfrage definiert im Wesentlichen ein Graph Pattern (Muster), das auf dem RDF-Graph *G*, auf dem die Anfrage operiert, ausgewertet wird. Dazu wird überprüft, ob die Variablen im Graph Pattern durch Knoten aus *G* ersetzt werden können, sodass der resultierende Graph in *G* enthalten ist (Pattern Matching). Grundlage jedes Graph Patterns bilden die *Basic Graph Patterns* (BGP). Ein BGP

⁴ SPARQL ist ein rekursives Akronym und steht für *SPARQL Protocol and RDF Query Language*.

besteht aus einer endlichen Menge an *Triple-Patterns*, die mittels AND (.) verkettet werden. Ein Triple-Pattern ist ein RDF-Tripel, wobei Subjekt, Prädikat und Objekt mit einer Variablen (?var) belegt sein können (z.B. ?s :p ?o). Ein Graph Pattern lässt sich dann rekursiv definieren:

- Ein *Basic Graph Pattern* ist ein Graph Pattern.
- Sind *P* und *P'* Graph Pattern, dann sind auch $\{P\} . \{P'\}$, *P* UNION *P'* und *P* OPTIONAL *P'* Graph Pattern.
- Ist *P* ein Graph Pattern und *R* eine Filter-Bedingung, dann ist auch *P* FILTER (*R*) ein Graph Pattern.

Mit Hilfe des FILTER-Operators lassen sich die Werte von Variablen im Graph Pattern beschränken und der OPTIONAL-Operator erlaubt das optionale Hinzufügen von Informationen zum Ergebnis einer Anfrage. Sollten die gewünschten Informationen nicht vorhanden sein, so bleiben die entsprechenden Variablen im Ergebnis *ungebunden*, d.h. es wird ihnen kein Wert zugeordnet. Mit Hilfe des UNION-Operators lassen sich zwei alternative Graph Patterns in einer Anfrage definieren. Ein Anfrage-Ergebnis muss dann mindestens eines der beiden Patterns erfüllen. Darüber hinaus gibt es in SPARQL auch einen GRAPH-Operator, der die Referenzierung mehrerer RDF-Graphen in einer Anfrage ermöglicht. Im Folgenden beschränken wir uns allerdings auf Anfragen, die sich nur auf einen RDF-Graph beziehen.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?person foaf:knows ex:Peter .
    ?person foaf:age ?age
    FILTER (?age >= 18)
    OPTIONAL {
        ?person foaf:mbox ?mb
    }
}
```

Abbildung 2: SPARQL-Anfrage mit FILTER und OPTIONAL

Die Anfrage aus Abbildung 2 ermittelt alle Personen, die "Peter" kennen und mindestens 18 Jahre alt sind. Sollte außerdem eine Mailbox-Adresse bekannt sein, so wird diese zum Ergebnis der Anfrage hinzugefügt. Tabelle 1 zeigt das Ergebnis der Anfrage auf dem RDF-Graph aus Abbildung 1.

?person	?age	?mb
ex:John	27	
ex:Bob	32	bob@abc.de

Tabelle 1: Auswertung der Anfrage

2.3 MapReduce

MapReduce ist ein von Google im Jahr 2004 vorgestelltes Modell für nebenläufige Berechnungen auf sehr großen Datenmengen unter Einsatz eines Computer-Clusters [DG04]. Inspiriert wurde das Konzept durch die in der funktionalen Programmierung häufig verwendeten Funktionen *map* und *reduce*. Ausgangspunkt für die Entwicklung war die Erkenntnis, dass viele Berechnungen bei Google zwar konzeptuell relativ einfach sind, jedoch zumeist auf sehr großen Datensätzen ausgeführt werden müssen. Eine parallelisierte Ausführung ist daher oftmals unerlässlich, weshalb selbst einfache Berechnungen eine komplexe Implementierung erforderten, um mit den Problemen einer parallelisierten Ausführung umgehen zu können. Bei der Entwicklung von MapReduce standen daher insbesondere eine gute Skalierbarkeit und Fehlertoleranz des Systems im Mittelpunkt, da bei großen Computer-Clustern immer mit Ausfällen gerechnet werden muss.

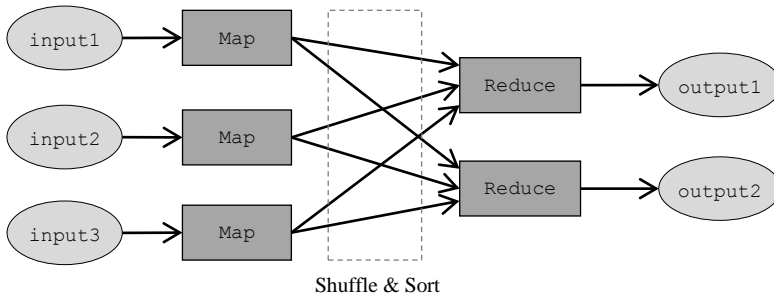


Abbildung 3: MapReduce-Datenfluss

Im Prinzip muss der Entwickler bei der Erstellung eines MapReduce-Jobs lediglich eine *Map*- und eine *Reduce*-Funktion implementieren, die vom Framework parallelisiert auf den Eingabe-Daten ausgeführt werden. Abbildung 3 zeigt den schematischen Ablauf eines MapReduce-Jobs mit drei Mappern und zwei Reducern. Technisch gesehen berechnet ein MapReduce-Job aus einer Liste von Schlüssel-Wert-Paaren (Eingabe) eine neue Liste von Schlüssel-Wert-Paaren (Ausgabe):

$$[(k_1, v_1), \dots, (k_n, v_n)] \mapsto [(k'_1, v'_1), \dots, (k'_m, v'_m)]$$

Beispiel. Angenommen es soll für eine Menge von Dokumenten berechnet werden, welche Wörter mit welcher Häufigkeit darin vorkommen. Eine Vorverarbeitung habe bereits eine Liste an Paaren $(docID, word)$ ergeben, wobei *docID* eine Referenz auf ein Dokument und *word* ein Wort aus dem entsprechenden Dokument repräsentieren. Diese Vorverarbeitung lässt sich ebenfalls mit Hilfe eines MapReduce-Jobs berechnen. Ein mögliches Ergebnis der Auswertung könnte dann so aussehen:

$$[(doc1, Peter), (doc1, Bob), (doc2, Peter), (doc2, Sarah)] \mapsto [(Peter, 2), (Bob, 1), (Sarah, 1)]$$

Eine wichtige Eigenschaft des MapReduce-Modells ist das sogenannte *Lokalitätsprinzip*. Um das Netzwerk zu entlasten wird dabei ausgenutzt, dass die Daten verteilt auf den Computern im Cluster abgespeichert sind. Das System versucht die Mapper so auf die Rechner zu verteilen, dass möglichst viele Daten lokal gelesen werden können und nicht über das Netzwerk übertragen werden müssen.

2.4 Pig Latin

Pig Latin ist eine von Yahoo! entworfene Sprache zur Analyse von großen Datenmengen [OI08], die für den Einsatz im Hadoop Framework entwickelt wurde, einer Open-Source Implementierung von Google's MapReduce. Die Implementierung von Pig Latin für Hadoop, *Pig*, übersetzt ein Pig Latin-Programm in eine Folge von MapReduce-Jobs und ist mittlerweile ein offizielles Subprojekt von Hadoop.

Datenmodell. Pig Latin besitzt ein vollständig geschachteltes Datenmodell und erlaubt dem Entwickler damit eine größere Flexibilität als die von der ersten Normalform vorgeschriebenen flachen Tabellen von relationalen Datenbanken. Das Datenmodell von Pig Latin kennt vier verschiedene Typen:

- *Atom*: Ein Atom beinhaltet einen einfachen, atomaren Wert wie eine Zeichenkette oder eine Zahl. Beispiel: 'Sarah' oder 24
- *Tupel*: Ein Tupel besteht aus einer Sequenz von *Feldern*, wobei jedes Feld einen beliebigen Datentyp besitzen kann. Jedem Feld in einem Tupel kann zudem ein *Name* (Alias) zugewiesen werden, über den das Feld referenziert werden kann.
Beispiel: ('John', 'Doe') mit Alias-Namen (Vorname, Nachname)
- *Bag*: Eine Bag besteht aus einer Kollektion von Tupeln, wobei ein Tupel auch mehrfach vorkommen darf. Darüber hinaus müssen die Schemata der Tupel nicht übereinstimmen, d.h. die Tupel können eine unterschiedliche Anzahl von Feldern mit unterschiedlichen Typen aufweisen.
Beispiel: $\left\{ \begin{array}{l} ('Bob', 'Sarah') \\ (('Peter', ('likes', 'football')) \end{array} \right\}$
- *Map*: Eine *Map* beinhaltet eine Kollektion von Datenelementen. Jedes Element kann dabei über einen zugeordneten Schlüssel referenziert werden.
Beispiel: $\left[\begin{array}{l} 'name' \rightarrow 'John' \\ 'knows' \rightarrow \left\{ \begin{array}{l} ('Sarah') \\ ('Bob') \end{array} \right\} \end{array} \right]$

Operatoren. Ein Pig Latin-Programm besteht aus einer Sequenz von Schritten, wobei jeder Schritt einer einzelnen Daten-Transformation entspricht. Da Pig Latin für die Bearbeitung von großen Datenmengen mit Hadoop entwickelt wurde, müssen die Operatoren gut *parallelisierbar* sein. Daher wurden konsequenterweise nur solche Operatoren aufgenommen, die sich in eine Folge von MapReduce-Jobs übersetzen und damit parallel ausführen lassen. Im Folgenden werden die für die Übersetzung wichtigsten Operatoren in aller Kürze vorgestellt. Für eine genauere Darstellung sei auf die offizielle Dokumentation von Pig Latin [Ap10] verwiesen.

- **LOAD**: Für die Bearbeitung mit Pig Latin müssen die Daten deserialisiert und in das Datenmodell von Pig Latin überführt werden. Hierfür kann eine *User Defined Function* (UDF) implementiert werden, die vom LOAD-Operator verwendet werden soll und das tupelweise Laden beliebiger Daten ermöglicht.
Beispiel: `persons = LOAD 'file' USING myLoad() AS (name,age,city);`

- **FOREACH:** Mit Hilfe des FOREACH-Operators lässt sich eine Verarbeitung auf jedes Tupel in einer Bag anwenden. Insbesondere lassen sich damit Felder eines Tupels entfernen oder neue Felder hinzufügen.

Beispiel: `result1 = FOREACH persons GENERATE
name, age>=18? 'adult':'minor' AS class;`

persons			result1	
name	age	city	name	class
Sarah	17	Freiburg	Sarah	minor
Bob	32	Berlin	Bob	adult

- **FILTER:** Der FILTER-Operator ermöglicht das Entfernen ungewollter Tupel aus einer Bag. Dazu wird die Bedingung auf alle Tupel in der Bag angewendet.

Beispiel: `result2 = FILTER persons BY age>=18;`

result2		
name	age	city
Bob	32	Berlin

- **[OUTER] JOIN:** Equi-Joins lassen sich in Pig Latin mit Hilfe des JOIN-Operators ausdrücken. Eine Besonderheit von Pig Latin ist darüber hinaus, dass sich ein JOIN auch auf mehr als zwei Relationen beziehen kann (*Multi-Join*). Über die Schlüsselwörter LEFT OUTER bzw. RIGHT OUTER können auch Outer Joins in Pig Latin realisiert werden.

Beispiel: `result3 = JOIN result1 BY name LEFT OUTER,
result2 BY name;`

result3				
result1:: name	result1:: class	result2:: name	result2:: age	result2:: city
Bob	adult	Bob	32	Berlin
Sarah	minor			

- **UNION:** Zwei oder mehr Bags können mit Hilfe des UNION-Operators zu einer Bag zusammengeführt werden, wobei Duplikate (mehrfach vorkommende Tupel) erlaubt sind. Im Gegensatz zu relationalen Datenbanken müssen die Tupel dabei nicht das gleiche Schema und insbesondere nicht die gleiche Anzahl an Feldern besitzen. Im Regelfall ist es allerdings nicht besonders empfehlenswert, Bags mit unterschiedlichen Schemata zu vereinigen, da die Schema-Informationen (speziell die Alias-Namen für die Felder der Tupel) dabei verloren gehen.

3 Übersetzung von SPARQL nach Pig Latin

Die direkte Übersetzung einer SPARQL-Anfrage in ein Pig Latin-Programm wäre aufgrund der komplexen Syntax und der datenorientierten Struktur von SPARQL äußerst schwierig. Deshalb wird die Anfrage zunächst nach dem offiziellen Schema des W3C in einen SPARQL Algebra-Baum überführt und anschließend in ein Pig Latin-Programm übersetzt. Abbildung 4 zeigt das grundlegende Konzept der Übersetzung, wie sie in dieser Arbeit vorgestellt wird. Die modulare Vorgehensweise bietet mehrere Vorteile, so können beispielsweise Optimierungen auf der Algebra-Ebene durchgeführt werden, ohne die Übersetzung der Algebra in ein Pig Latin-Programm verändern zu müssen.

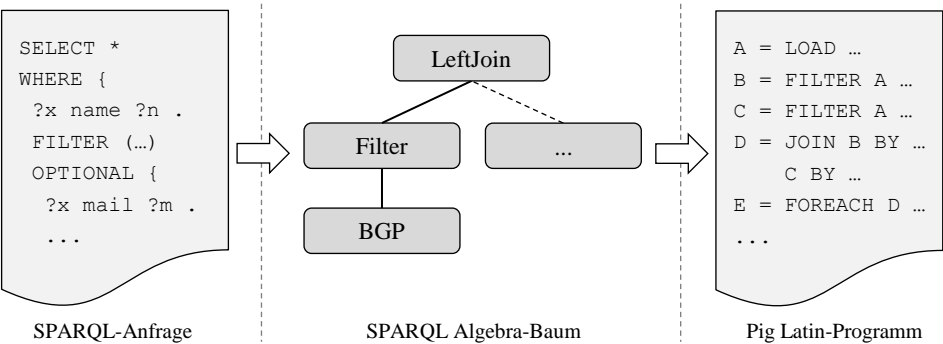


Abbildung 4: Schematischer Ablauf der Übersetzung

3.1 SPARQL Algebra

Zur Auswertung einer SPARQL-Anfrage wird die Anfrage zunächst in einen Ausdruck der SPARQL Algebra überführt, da die Semantik einer Anfrage auf der Ebene der SPARQL Algebra definiert ist. Ein solcher Ausdruck lässt sich in Form eines *Algebra-Baums* repräsentieren. Tabelle 2 stellt die Operatoren der SPARQL Algebra ihren Entsprechungen in der SPARQL Syntax gegenüber.

Algebra	SPARQL Syntax
<i>BGP</i>	Menge von Triple-Patterns (verkettet über Punkt-Symbol)
<i>Join</i>	Verknüpfung zweier Gruppen ($\{...\}.\{...\}$)
<i>LeftJoin</i>	OPTIONAL
<i>Filter</i>	FILTER
<i>Union</i>	UNION
<i>Graph</i>	GRAPH

Tabelle 2: Zusammenhang zwischen SPARQL Algebra und Syntax

3.2 Abbildung der RDF-Daten in das Datenmodell von Pig

Ein RDF-Datensatz besteht im Wesentlichen aus einer Menge von RDF-Tripeln und ein RDF-Tripel setzt sich aus URIs, RDF-Literalen und Blank Nodes zusammen. URIs sind nach ihrer Syntax spezielle ASCII-Zeichenfolgen und lassen sich daher im Datenmodell von Pig als Atome repräsentieren (`<URI>`). RDF unterscheidet des Weiteren zwischen *einfachen* und *getypten* Literalen. Einfache Literale sind Unicode-Zeichenfolgen und können daher ebenfalls als Atome repräsentiert werden. Getypte Literale haben entweder einen zusätzlichen *Language-Tag* oder einen *Datentyp* und lassen sich durch einen zusammengesetzten Wert (`"literal"@lang` bzw. `"literal"^^datatype`) repräsentieren. Bei der Auswertung von arithmetischen Ausdrücken werden die Literale zur Laufzeit in den entsprechenden Typ umgewandelt. Hierfür wurden für die bei RDF gebräuchlichen Datentypen (z.B. `xsd:integer`) spezielle Umwandlungen definiert. Die RDF-Syntax macht keine näheren Angaben zur internen Struktur von Blank Nodes, sie müssen lediglich von URIs und Literalen unterscheidbar sein. Um dies zu gewährleisten, kann eine Darstellung der Form `_:nodeID` verwendet werden. Ein RDF-Tripel lässt sich somit als Tupel aus drei atomaren Feldern mit dem Schema (`s:chararray, p:chararray, o:chararray`) darstellen.

3.3 Übersetzung der SPARQL Algebra

Im Folgenden werden für die Operatoren der SPARQL Algebra Vorschriften zur Übersetzung in eine Folge von Pig Latin-Befehlen angegeben. Hierfür muss zunächst die benötigte Terminologie eingeführt werden, die analog zu [PAG09] definiert wird: Sei V die unendliche Menge an Anfrage-Variablen und T die Menge der gültigen RDF-Terme (URIs, RDF-Literale, Blank Nodes).

Definition 1. Ein (*Solution*) *Mapping* μ ist eine partielle Funktion $\mu: V \rightarrow T$. Die Domain von μ , $dom(\mu)$, ist die Teilmenge von V , wo μ definiert ist. Im Folgenden wird ein Solution Mapping umgangssprachlich auch als *Ergebnis* bezeichnet.

Definition 2. Zwei Solution Mappings μ_1 und μ_2 sind *kompatibel*, wenn für alle Variablen $?X \in dom(\mu_1) \cap dom(\mu_2)$ gilt, dass $\mu_1(?X) = \mu_2(?X)$. Es folgt, dass $\mu_1 \cup \mu_2$ wieder ein Solution Mapping ergibt und zwei Mappings mit disjunkten Domains immer kompatibel sind.

Probleme bei der Auswertung von SPARQL mit Pig Latin bereiten *ungebundene* Variablen, die durch die Anwendung des OPTIONAL-Operators entstehen können. Im Gegensatz zu einem NULL-Wert in der relationalen Algebra führt eine ungebundene Variable in SPARQL bei der Auswertung eines Joins nicht dazu, dass das entsprechende Tupel verworfen wird [Cy05]. Da sich Joins in Pig Latin an der relationalen Algebra orientieren und ungebundene Variablen als NULL-Werte in Pig Latin dargestellt werden, führt dies zu unterschiedlichen Ergebnissen bei der Auswertung. Aus diesem Grund betrachten wir bei der Übersetzung nach Pig Latin *schwach wohlgeformte* Graph Pattern, die in Anlehnung an *wohlgeformte* Graph Pattern nach Pérez et al. [PAG09] definiert werden aber weniger restriktiv sind. Insbesondere sind die meisten Anfragen aus der offiziellen SPARQL Dokumentation [PS08] schwach wohlgeformt.

Definition 3. Ein Graph Pattern P ist *schwach wohlgeformt*, wenn es kein GRAPH enthält, UNION nicht in einem anderen Operator enthalten ist und für jedes Sub-Pattern $P' = (P_1 \text{ OPTIONAL } P_2)$ von P und jede Variable $?X$ aus P gilt: Falls $?X$ sowohl in P_2 als auch außerhalb von P' vorkommt, dann kommt sie auch in P_1 vor.

Ist das Graph Pattern einer Anfrage schwach wohlgeformt, so treten keine Joins über NULL-Werte auf, da dies nur der Fall ist, falls eine Variable in P_2 und außerhalb von P' vorkommt aber nicht in P_1 oder nach einem UNION noch weitere Operatoren folgen. Anfragen, die nicht schwach wohlgeformt sind und somit bei der Auswertung zu einem NULL-Join führen, werden von unserem Übersetzer erkannt.

Basic Graph Pattern (BGP). BGPs bilden die Grundlage jeder SPARQL-Anfrage und werden direkt auf dem entsprechenden RDF-Graph ausgewertet. Sie liefern als Ergebnis eine Menge von Solution Mappings, die als Eingabe für die weiteren Operatoren dienen. Genauer gesagt handelt es sich dabei um eine *Multi-Menge*, da ein Solution Mapping mehrfach vorkommen kann.

Beispiel: $P_1 = \text{BGP} (?A \text{ knows Bob} . ?A \text{ age ?B} . ?A \text{ mbox ?C})$

Das BGP ermittelt alle Personen, die Bob kennen und für die sowohl das Alter als auch eine Mailbox-Adresse bekannt sind. Solution Mappings lassen sich in Pig in Form einer Relation (flache Bag) repräsentieren. Jedes Tupel der Relation entspricht einem Solution Mapping und die Felder des Tupels entsprechen den Werten für die Variablen. Das Schema der Relation bilden die Namen der Variablen ohne führendes Fragezeichen, da diese bei Alias-Namen in Pig nicht erlaubt sind. Abbildung 5 zeigt die Übersetzung von P_1 in eine Folge von Pig Latin-Befehlen.

```
graph = LOAD 'pathToFile' USING rdfLoader() AS (s,p,o) ; (1)

t1 = FILTER graph BY p == 'knows' AND o == 'Bob' ; (2)
t2 = FILTER graph BY p == 'age' ;
t3 = FILTER graph BY p == 'mbox' ;

j1 = JOIN t1 BY s, t2 BY s ; (3)
j2 = JOIN j1 BY t1::s, t3 BY s;

P1 = FOREACH j2 GENERATE (4)
    t1::s AS A, t2::o AS B, t3::o AS C ;
```

Abbildung 5: Übersetzung eines BGPs

- (1) Das Laden der Daten erfolgt mit Hilfe des LOAD-Operators. Hierfür muss eine spezielle *Loader-UDF* implementiert werden. Anschließend stehen die Daten im Format aus Abschnitt 3.1 zur Verfügung.
- (2) Für jedes Triple-Pattern im BGP wird ein FILTER benötigt, der diejenigen RDF-Tripel selektiert, die das Triple-Pattern erfüllen (Pattern Matching).
- (3) Die Ergebnisse der FILTER werden dann sukzessive mit Hilfe des JOIN-Operators verknüpft. In jedem Schritt wird dabei ein weiteres Triple-Pattern zur berechneten Lösung hinzugenommen. Besteht das BGP aus n Triple-Patterns, so sind folglich $n-1$ Joins erforderlich. Das Prädikat des Joins ergibt sich jeweils aus

den gemeinsamen Variablen der beiden Argumente. Der Join verknüpft damit die kompatiblen Solution Mappings der beiden Argumente und erzeugt daraus neue Solution Mappings. Sollte es keine gemeinsamen Variablen geben, so muss das Kreuzprodukt der beiden Argumente berechnet werden.

- (4) Durch ein abschließendes FOREACH werden die überflüssigen Spalten der Relation mit den berechneten Solution Mappings entfernt und das Schema der Relation an die Variablenamen angepasst.

Filter. Der Filter-Operator der SPARQL Algebra dient dazu, aus einer Multi-Menge an Solutions Mappings diejenigen Mappings zu entfernen, welche die Filter-Bedingung nicht erfüllen.

Beispiel: `P2 = Filter(?B >= 30 && ?B <= 40, P1)`

Aus den Ergebnissen (Solution Mappings) für das Pattern P1 sollen diejenigen Personen entfernt werden, die jünger als 30 oder älter als 40 Jahre sind. Ein Filter lässt sich in Pig Latin mit Hilfe des FILTER-Befehls ausführen. Nicht alle Filter-Bedingungen in SPARQL lassen sich allerdings direkt in Pig Latin ausdrücken. So ist z.B. die Syntax von regulären Ausdrücken in SPARQL und Pig Latin verschieden⁵.

```
P2 = FILTER P1 BY (B >= 30 AND B <= 40) ;
```

Abbildung 6: Übersetzung eines Filters

Join. Der Join-Operator der SPARQL Algebra bekommt als Eingabe zwei Multi-Mengen von Solution Mappings. Er kombiniert die kompatiblen Solution Mappings aus beiden Mengen und erzeugt so eine neue Multi-Menge an Solution Mappings für das zusammengesetzte Pattern.

Beispiel: `P3 = Join(BGP(?A knows ?B), BGP(?A age ?C . ?B age ?C))`

Das linke Pattern (*P*) liefert alle Personen, die eine andere Person kennen und das rechte Pattern (*P'*) liefert alle Personen-Paare, die das gleiche Alter haben. Über den Join-Operator werden die beiden Mengen von Solution Mappings zu einer Menge verknüpft. Das Ergebnis der Anfrage sind somit alle Paare von Personen, die sich kennen und gleich alt sind. Ein Join lässt sich in Pig Latin analog zum BGP mit Hilfe des JOIN-Befehls realisieren. Das Prädikat des Joins ergibt sich auch hier aus den gemeinsamen Variablen der beiden Eingabe-Relationen (Multi-Mengen von Solution Mappings). Sollte es keine gemeinsamen Variablen geben, so muss das Kreuzprodukt der beiden Relationen berechnet werden. Abschließend werden mit FOREACH die überflüssigen Spalten entfernt und das Schema der Ergebnis-Relation angepasst.

```
j1 = JOIN BGP1 BY (A,B), BGP2 BY (A,B) ;
P3 = FOREACH j1 GENERATE
    BGP1::A AS A, BGP1::B AS B, BGP2::C AS C ;
```

Abbildung 7: Übersetzung eines Joins

⁵ SPARQL unterstützt reguläre Ausdrücke wie in XPath 2.0 oder XQuery 1.0 während Pig Latin die umfangreicheren regulären Ausdrücke von Java unterstützt (vgl. [OI08]).

LeftJoin. Mit Hilfe des LeftJoin-Operators können zusätzliche Informationen zum Ergebnis hinzugenommen werden, falls diese vorhanden sind. Konzeptionell entspricht der LeftJoin damit einem klassischen Left-Outer Join. Der LeftJoin kann auch eine Filter-Bedingung beinhalten, die als Bedingung für den Outer Join interpretiert werden kann. Eine Darstellung der Übersetzung eines LeftJoins mit Filter ist im Rahmen dieses Beitrags nicht möglich. Hierfür sei auf die vollständige Ausarbeitung [Sc10] verwiesen.

Beispiel: $P4 = \text{LeftJoin}(\text{BGP}(\text{?A age ?B}), \text{BGP}(\text{?A mbox ?C}), \text{true})$

Enthält der LeftJoin keinen Filter, so wird dies wie in P4 durch eine Filter-Bedingung ausgedrückt, die immer erfüllt ist (true). Der LeftJoin aus P4 liefert alle Personen, für die das Alter bekannt ist. Sollte außerdem noch eine Mailbox-Adresse bekannt sein, so wird auch diese zum Ergebnis hinzugenommen. Ohne Filter lässt sich der LeftJoin als normaler OUTER JOIN in Pig Latin auf den gemeinsamen Variablen der beiden Eingabe-Relationen realisieren. Gibt es keine gemeinsamen Variablen, so muss auch hier das Kreuzprodukt der beiden Relationen berechnet werden. Ein abschließendes FOREACH entfernt die überflüssigen Spalten und passt das Schema der Ergebnis-Relation an.

```
lj = JOIN BGP1 BY A LEFT OUTER, BGP2 BY A ;
P4 = FOREACH lj GENERATE
      BGP1::A AS A, BGP1::B AS B, BGP2::C AS C ;
```

Abbildung 8: Übersetzung eines LeftJoins ohne Filter

Union. Der Union-Operator der SPARQL Algebra fasst zwei Multi-Mengen von Solution Mappings zu einer Multi-Menge zusammen. Damit lassen sich folglich die Ergebnisse von zwei Graph Patterns vereinigen.

Beispiel: $P5 = \text{Union}(\text{BGP}(\text{?A knows Bob . ?A mbox ?B}), \text{BGP}(\text{?A knows John}))$

Das linke Pattern (P) liefert alle Personen, die Bob kennen und deren Mailbox-Adresse bekannt ist. Das rechte Pattern (P') hingegen liefert alle Personen, die John kennen, unabhängig von einer Mailbox-Adresse. Obwohl der Union-Operator zunächst relativ unproblematisch wirkt, ist die Übersetzung mit einigen Problemen verbunden. Das liegt daran, dass für zwei Mappings $\mu \in P$ und $\mu' \in P'$ gelten kann, dass $\text{dom}(\mu) \neq \text{dom}(\mu')$, wie es beispielsweise bei P5 der Fall ist. In diesem Fall müssen zunächst die Schemata der beiden Relationen mit Hilfe von FOREACH aneinander angepasst werden, indem für ungebundene Variablen Null-Werte eingeführt werden. Andernfalls gehen die Schemata der beiden Relationen verloren, da die Ergebnis-Relation weder das Schema der einen noch das Schema der anderen Relation übernehmen kann. Da die Variablen-Namen der Solution Mappings allerdings im Schema der Relation definiert sind, wäre dies äußerst problematisch. Im Beispiel von P5 muss z.B. zunächst das Schema der rechten Relation an das Schema der linken Relation angepasst werden (Abbildung 9).

```
BGP2 = FOREACH BGP2 GENERATE A, null AS B ;
P5    = UNION BGP1, BGP2 ;
```

Abbildung 9: Übersetzung eines Unions

Eine vollständige Darstellung der entwickelten Übersetzung ist im Rahmen dieses Beitrags leider nicht möglich. Der interessierte Leser sei hierfür auf die vollständige Darstellung in [Sc10] verwiesen.

3.4 Optimierungen

Die Optimierung von SPARQL-Anfragen ist Gegenstand aktueller Forschung [HH07, St08, SML10]. Im Folgenden werden einige von uns untersuchte Optimierungsstrategien für die entwickelte Übersetzung kurz dargestellt. Bei den ersten Evaluationen hat sich gezeigt, dass eine Optimierung vor allem die Reduktion des Datenaufkommens (Input/Output, I/O) zum Ziel haben sollte. Das beinhaltet zum einen die Daten, welche innerhalb eines MapReduce-Jobs von den Mappern zu den Reducern übertragen werden und zum anderen die Daten, welche zwischen zwei MapReduce-Jobs in das verteilte Dateisystem von Hadoop, *HDFS*, übernommen werden müssen.

- (1) **SPARQL Algebra.** Zur Reduktion der erzeugten Zwischenergebnisse einer Anfrage wurden Optimierungen des *Filter*- und des *BGP*-Operators betrachtet. Ziel dieser Optimierungen ist die möglichst frühzeitige Auswertung von Filtern sowie die Neuordnung der Triple-Patterns in einem BGP entsprechend ihrer Selektivität [St08]. Dabei werden die Triple-Patterns nach der Anzahl und Position ihrer Variablen geordnet, da ein Triple-Pattern mit zwei Variablen als weniger selektiv angesehen wird wie ein Triple-Pattern mit nur einer Variablen und Subjekte im Allgemeinen selektiver sind als Prädikate.
- (2) **Übersetzung der Algebra.** Es hat sich als äußerst wirksam erwiesen, unnötige Daten so früh wie möglich aus einer Relation zu entfernen (*"Project early and often"*). Darüber hinaus spielt die effiziente Auswertung von Joins [NW09] eine entscheidende Rolle. Hier hat sich insbesondere die Verwendung von Multi-Joins in Pig Latin bei bestimmten Anfragen bewährt, da dadurch die Anzahl der benötigten Joins reduziert werden kann. Ein Multi-Join ist dann möglich, wenn sich mehrere aufeinander folgende Joins auf die gleichen Variablen beziehen. Betrachten wir folgendes Beispiel: Angenommen es sollen drei Relationen (A, B, C) über die Variable ?x zusammengeführt werden. Normalerweise sind hierfür zwei Joins und somit zwei MapReduce-Phasen erforderlich (1). In Pig Latin lassen sich die beiden Joins allerdings zu einem Multi-Join und somit auch zu einer MapReduce-Phase zusammenfassen (2).

$\begin{array}{l} j1 = \text{JOIN } A \text{ BY } x, B \text{ BY } x ; \\ j2 = \text{JOIN } j1 \text{ BY } A::x, C \text{ BY } x ; \end{array} \quad (1)$ <hr style="width: 100%;"/> $j1 = \text{JOIN } A \text{ BY } x, B \text{ BY } x, C \text{ BY } x ; \quad (2)$
--

Abbildung 10: Multi-Join in Pig Latin

Bei der Übersetzung eines BGPs in eine Folge von Joins in Pig Latin wird die Reihenfolge der Triple-Patterns daher so angepasst, dass möglichst viele Joins zu einem Multi-Join zusammengefasst werden können, auch wenn dadurch das Selektivitäts-Kriterium aus (1) verletzt wird.

- (3) **Datenmodell.** Betrachtet man eine typische SPARQL-Anfrage genauer, so sind die Prädikate in den Triple-Patterns in den meisten Fällen gebunden. Eine *vertikale Partitionierung* der RDF-Daten nach Prädikaten [Ab07] reduziert daher oftmals die Menge an RDF-Tripeln, die zur Auswertung einer Anfrage geladen werden müssen. Bei einem ungebundenen Prädikat muss allerdings weiterhin der komplette Datensatz geladen werden.

3.5 Übersetzung einer Beispiel-Anfrage

Im Folgenden wird die Übersetzung einer SPARQL-Anfrage in ein entsprechendes Pig Latin-Programm anhand eines kleinen Beispiels dargestellt. Abbildung 11 zeigt eine SPARQL-Anfrage mit dem entsprechenden Algebra-Baum. Der Baum wird von unten nach oben traversiert und in eine Folge von Pig Latin-Befehlen übersetzt (Abbildung 12), wobei eine vertikale Partitionierung der Daten unterstellt wird. Dabei werden unnötige Daten mit Hilfe von FOREACH so früh wie möglich entfernt, was die Menge an Daten reduziert, die über das Netzwerk übertragen werden müssen.

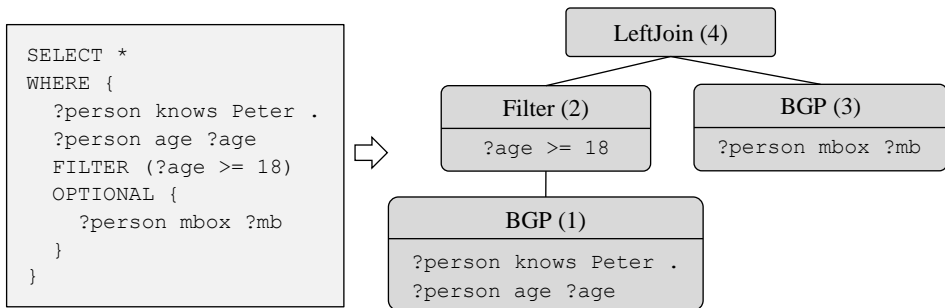


Abbildung 11: SPARQL Algebra-Baum

```

knows = LOAD 'pathToFile/knows' USING rdfLoader() AS (s,o) ;           (1)
age    = LOAD 'pathToFile/age' USING rdfLoader() AS (s,o) ;
f1     = FILTER knows BY o == 'Peter' ;
t1     = FOREACH f1 GENERATE s AS person ;
t2     = FOREACH age GENERATE s AS person, o AS age ;
j1     = JOIN t1 BY person, t2 BY person ;
BGP1   = FOREACH j1 GENERATE t1::person AS person, t2::age AS age ;

F1     = FILTER BGP1 BY age >= 18 ;                                     (2)

mbox   = LOAD 'pathToFile/mbox' USING rdfLoader() AS (s,o) ;           (3)
BGP2   = FOREACH mbox GENERATE s AS person, o AS mb ;

lj     = JOIN F1 BY person LEFT OUTER, BGP2 BY person ;                 (4)
LJ1    = FOREACH lj GENERATE
  F1::person AS person, F1::age AS age, BGP2::mb AS mb ;
STORE LJ1 INTO 'pathToOutput' USING resultWriter();
  
```

Abbildung 12: Übersetzung des Algebra-Baums

4 Evaluation

Für die Evaluation wurden zehn Dell PowerEdge R200 Server mit jeweils einem Dual Core Intel Xeon E3120 3,16 GHz Prozessor, 4 GB DDR2 800 MHz Arbeitsspeicher, 1 TB SATA-Festplatte mit 7200 U/min und einem Dual Port Gigabit Ethernet Adapter verwendet. Die Server wurden über einen 3Com Baseline Switch 2824 zu einem Gigabit-Netzwerk zusammengeschaltet und auf den Servern wurde Ubuntu 9.10 Server (x86_64), Java in der Version 1.6.0_15 und Cludera's Distribution for Hadoop 3 (CDH3)⁶ installiert. Zum Zeitpunkt der Evaluation beinhaltete CDH3 unter anderem Hadoop in der Version 0.20.2 sowie Pig in der Version 0.5.0. Insgesamt standen knapp 8 TB an Festplattenspeicher zur Verfügung, was bei einem Replikationsfaktor von drei des verteilten Dateisystems von Hadoop (HDFS) ungefähr 2,5 TB an Nutzdaten entspricht.

Als Kennzahlen wurden neben der *Ausführungszeit* einer Anfrage auch die Menge an Daten ermittelt, die aus dem HDFS gelesen (*HDFS Bytes Read*), in das HDFS geschrieben (*HDFS Bytes Written*) sowie von den Mappern zu den Reducern übertragen (*Reduce Shuffle Bytes*) wurden. Für die Evaluation wurde der SP²Bench [Sc09] verwendet, ein SPARQL spezifischer Performance Benchmark. Der Datengenerator des SP²Bench erlaubt das Erzeugen beliebig großer RDF-Dateien auf der Grundlage der DBLP-Bibliothek von Michael Ley [Le10]. Er berücksichtigt dabei insbesondere die charakteristischen Eigenschaften und Verteilungen eines DBLP-Datensatzes und liefert somit ein realistisches Datenmodell. Im Folgenden wird die Auswertung von zwei charakteristischen Anfragen des SP²Bench präsentiert. Weitere Evaluationsergebnisse finden sich in der vollständigen Ausarbeitung [Sc10].

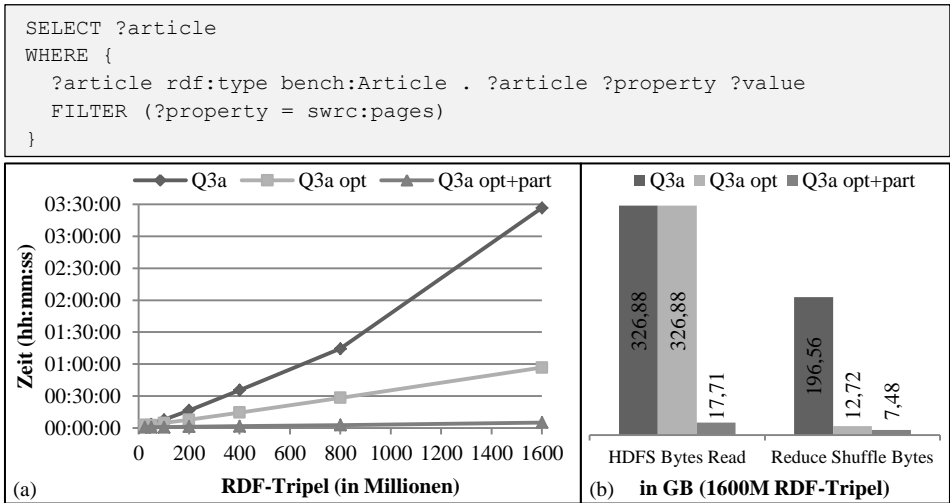


Abbildung 13: Auswertung von Q3a

Abbildung 13 zeigt die Auswertung von Q3a des SP²Bench. Die Anfrage benötigt zur Auswertung zwar nur einen Join, dieser berechnet aber sehr viele Zwischenergebnisse,

⁶ siehe [<http://www.cludera.com/hadoop/>]

da alle RDF-Tripel in der Eingabe das zweite Triple-Pattern erfüllen. Das spiegelt sich auch in den erzeugten Reduce Shuffle Bytes wieder, die bei der Berechnung des Joins anfallen. Da die Filter-Variable `?property` allerdings nicht in der Ausgabe enthalten sein soll, lässt sich die Anfrage auf Algebra-Ebene durch eine Filter-Substitution optimieren. Dabei wird die Variable durch ihren entsprechenden Filter-Wert ersetzt, wodurch der ursprüngliche Filter überflüssig wird. Durch diese Optimierung lässt sich die Ausführungszeit der Anfrage (a) beim größten Datensatz um über 70% verringern (Q3a opt), was auf eine signifikante Reduktion der Reduce Shuffle Bytes (b) zurückzuführen ist. Ein positiver Nebeneffekt der Optimierung ist die Eliminierung des ungebundenen Prädikats im zweiten Triple-Pattern, wovon insbesondere die Auswertung auf einem vertikal partitionierten Datensatz profitiert (Q3a opt+part). Da dadurch nur noch die beiden Prädikate `rdf:type` und `swrc:pages` betrachtet werden müssen, wird die Menge der Daten, die aus dem HDFS gelesen werden, deutlich reduziert. Durch Anwendung der Filter-Optimierung und der vertikalen Partitionierung lässt sich die Ausführungszeit der Anfrage auf dem größten Datensatz somit insgesamt um über 97% reduzieren.

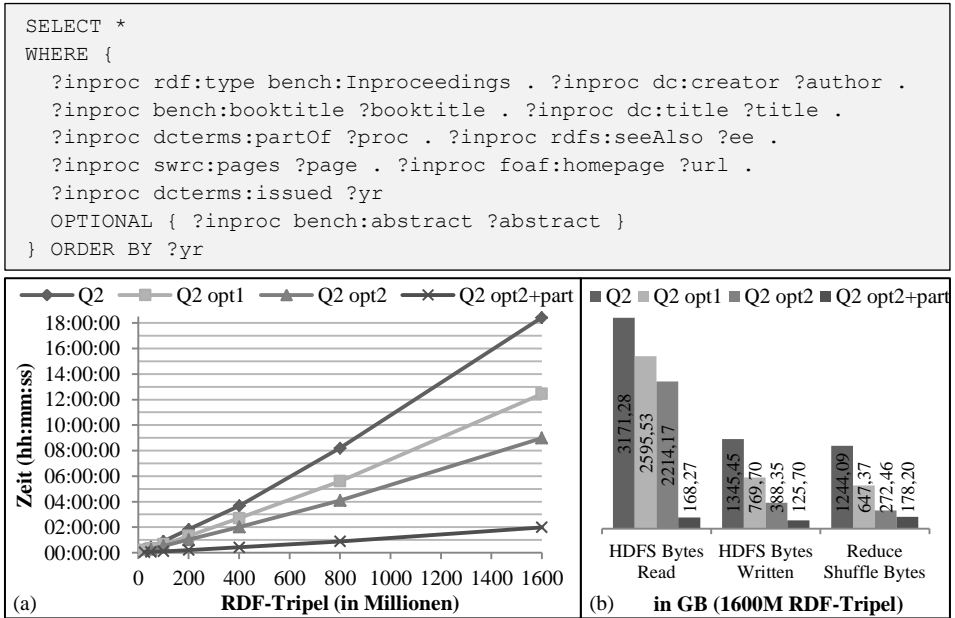


Abbildung 14: Auswertung von Q2

Abbildung 14 zeigt die Auswertung von Q2 des SP²Bench. Dabei handelt es sich um eine komplexe Anfrage, die viele Joins erfordert und zudem ein OPTIONAL enthält. Die Ergebnisse sollen darüber hinaus in sortierter Reihenfolge ausgegeben werden. Das linke BGP der Anfrage besteht aus neun Triple-Patterns, weshalb zur nativen Auswertung insgesamt acht Joins und ein Outer Join erforderlich sind. Durch das frühzeitige Entfernen unnötiger Spalten (Projektion, Q2 opt1) lässt sich die Ausführungszeit (a) der Anfrage auf dem größten Datensatz bereits um mehr als 30% verkürzen, was sich auch in den ermittelten Daten-Kennzahlen (b) niederschlägt. Darüber hinaus lässt sich bei der Übersetzung von Q2 die Multi-Join-Fähigkeit von Pig Latin ausnutzen. Da sich alle acht

Joins auf die Variable `?inproc` beziehen, können sie in Pig Latin zu einem einzigen Join zusammengefasst werden (Q2 opt2). Dadurch reduziert sich auch die benötigte Anzahl an MapReduce-Jobs zur Auswertung von Q2 von ursprünglich zwölf bei iterierten Joins auf fünf bei Verwendung eines Multi-Joins. Da alle Prädikate in der Anfrage gebunden sind, wirkt sich auch eine vertikale Partitionierung der Daten nach Prädikaten (Q2 opt2+part) vorteilhaft aus, was sich in den Daten-Kennzahlen ganz deutlich zeigt. Durch die Anwendung aller Optimierungen lässt sich die Ausführungszeit der Anfrage auf dem größten Datensatz folglich insgesamt um fast 90% reduzieren.

4.1 Erkenntnisse der Evaluation

Durch die Evaluation konnte die ursprüngliche Vermutung bestätigt werden, dass die Ausführungszeit einer Anfrage stark mit dem erzeugten Datenaufkommen korreliert, was in den betrachteten Kennzahlen deutlich zum Ausdruck kommt. Darüber hinaus konnte auch die Wirksamkeit der untersuchten Optimierungen bestätigt werden, die einen großen Einfluss auf die Ausführungszeiten hatten, was primär auf eine Reduktion des erzeugten Datenaufkommens zurückgeführt werden konnte. Ermutigend für zukünftige Weiterentwicklungen des gewählten Ansatzes zur Auswertung von SPARQL-Anfragen sind auch die beobachteten, linearen Skalierungen der Ausführungszeiten sowie die auf Anheb erreichten Datensatz-Größen von bis zu 1600 Millionen RDF-Tripeln, selbst ohne vertikale Partitionierung der Daten. Es ist anzunehmen, dass durch die vertikale Partitionierung und eine feinere Optimierung des Hadoop-Clusters dieser Wert noch gesteigert werden kann, von einer Vergrößerung des Clusters ganz abgesehen.

In [Hu10] wird ebenfalls die Auswertung von SPARQL-Anfragen mit Hadoop betrachtet, wobei im Gegensatz zu unserem Ansatz eine Anfrage direkt in eine Folge von MapReduce-Jobs übersetzt wird. Dabei werden auch Evaluations-Ergebnisse für die SP²Bench-Anfragen Q1, Q2 und Q3a mit unterschiedlichen Datensatz-Größen gezeigt, die auf einem Hadoop-Cluster aus zehn Knoten erzielt wurden, das unserem Cluster sehr ähnlich ist. Ein Vergleich der Ergebnisse zeigt, dass die beiden Ansätze eine ähnliche Performance aufweisen, wobei unser Ansatz bei Q3a um bis zu 40% bessere Werte erzielt, was wahrscheinlich auf die Optimierung des enthaltenen Filters zurückzuführen ist. Das zeigt, dass unser Ansatz einer Übersetzung von SPARQL nach Pig Latin eine effiziente Auswertung ermöglicht, die mit der Performance einer direkten Auswertung in MapReduce mithalten kann und zudem von der schnellen Weiterentwicklung von Pig profitiert [Ga09].

5 Verwandte Arbeiten

Die Übersetzung von Anfrage-Sprachen in andere Sprach-Konstrukte ist eine übliche Vorgehensweise, insbesondere im Bereich der relationalen Algebra [Bo05, Cy05]. Da die Semantik von Pig Latin stark an der relationalen Algebra orientiert ist, treten bei der Übersetzung von SPARQL nach Pig Latin die gleichen Probleme mit NULL-Werten bei Joins auf, wie sie auch bei der Übersetzung von SPARQL in die relationale Algebra zu finden sind [Cy05]. In [MT08] wurde bereits von einer Übersetzung von SPARQL nach

Pig Latin berichtet, zu der allerdings keine genaueren Angaben gemacht wurden. Nach unserem Kenntnisstand ist die in diesem Beitrag beschriebene Übersetzung die erste vollständige und detaillierte Darstellung einer Übersetzung von SPARQL nach Pig Latin, die darüber hinaus auch effiziente Optimierungen betrachtet und mit einem SPARQL spezifischen Benchmark evaluiert wurde.

Die Auswertung von SPARQL-Anfragen spielt im Bereich des semantischen Webs eine wichtige Rolle. Sesame [BKH02], Jena [Mc01], RDF-3X [NW08] und 3store [HG03] sind in diesem Zusammenhang bekannte Beispiele für die Auswertung von SPARQL-Anfragen auf Einzelplatz-Systemen. Mit der Zunahme an verfügbaren semantischen Daten⁷ rückt auch die Auswertung von großen RDF-Datensätzen zunehmend in den Blickpunkt wissenschaftlicher Forschung. [HBF09] und [QL08] betrachten in diesem Zusammenhang die Auswertung von SPARQL-Anfragen auf mehreren verteilten RDF-Datensätzen. Die meisten Ansätze zur Verwaltung und Auswertung von sehr großen RDF-Datensätzen mit mehreren Milliarden RDF-Tripeln setzen auf den Einsatz von Computer-Clustern. [Hu10] und [MYL10] befassen sich ebenfalls mit der Auswertung von SPARQL-Anfragen in einem MapReduce-Cluster. Im Gegensatz zu dem von uns vorgestellten Ansatz wird eine SPARQL-Anfrage dabei direkt in eine Folge von MapReduce-Jobs übersetzt, wobei allerdings größtenteils nur Basic Graph Patterns unterstützt werden. Die von uns vorgestellte Übersetzung nach Pig Latin unterstützt hingegen alle Operatoren der SPARQL Algebra (mit Ausnahme des GRAPH-Operators) und profitiert zudem von Optimierungen und Weiterentwicklungen von Pig. SHARD [RS10] ist ein RDF-Triple-Store für Hadoop, der auch SPARQL-Anfragen unterstützt. Experimentelle Ergebnisse liegen allerdings nur für den allgemeinen LUBM-Benchmark [GPH05] vor, der wichtige Eigenschaften von SPARQL-Anfragen nicht berücksichtigt. Die Autoren machen hier leider auch keine genauen Angaben zum unterstützten Sprachumfang. SPIDER [Ch09] verwendet HBase zur Speicherung von RDF-Daten in Hadoop in Form von flachen Tabellen und unterstützt auch grundlegende SPARQL-Anfragen, wobei auch hier keine genaueren Angaben zum unterstützten Sprachumfang gemacht werden. In [RDA10] wird die Verwendung von UDFs zur Reduzierung der I/O-Kosten bei der Auswertung von analytischen Anfragen auf RDF-Graphen mit Pig Latin untersucht. Dabei wurde gezeigt, dass durch die Verwendung spezieller UDFs die I/O-Kosten in einigen Situationen gesenkt werden können, weshalb sich eine Übertragung des Ansatzes auf die hier vorgestellte Übersetzung als vorteilhaft erweisen könnte.

Neben dem Einsatz eines allgemeinen MapReduce-Clusters setzen einige Systeme auch auf spezialisierte Computer-Cluster. Virtuoso Cluster Edition [Er10] und Clustered TDB [Ow09] sind Cluster-Erweiterungen der bekannten Virtuoso und Jena RDF-Stores. 4store [HLS09] ist ein einsatzbereiter RDF-Store, bei dem das Cluster in Storage und Processing Nodes unterteilt wird. YARS2 [Ha07] setzt auf die Verwendung von Indexen zur Anfrage-Auswertung und MARVIN [Or10] verwendet einen Peer-to-Peer-Ansatz zur verteilten Berechnung von RDF Reasoning. Die Verwendung von spezialisierten Clustern hat allerdings den Nachteil, dass hierfür eine eigene Infrastruktur aufgebaut werden muss, wohingegen unser Ansatz auf der Verwendung eines allgemeinen Clusters beruht, das für verschiedene Zwecke verwendet werden kann.

⁷ siehe [<http://esw.w3.org/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics>]

6 Zusammenfassung

In diesem Beitrag wird ein neuer Ansatz zur effizienten Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen unter Verwendung des Hadoop MapReduce-Frameworks vorgestellt. Dazu wurde für schwach wohlgeformte SPARQL-Anfragen eine Übersetzung nach Pig Latin entwickelt und implementiert. Schwach wohlgeformte Anfragen sind ein ausdrucks mächtiges Fragment von SPARQL, die in der Praxis sehr häufig vorkommen. Es wurden für die Operatoren der SPARQL Algebra entsprechende Übersetzungsvorschriften entwickelt und eine Abbildung des RDF-Datenmodells in das Datenmodell von Pig definiert. Das resultierende Pig Latin-Programm wird von Pig, der Implementierung von Pig Latin für Hadoop, in eine Folge von MapReduce-Jobs überführt und verteilt auf einem Hadoop-Cluster ausgeführt. Die Evaluationsergebnisse haben gezeigt, dass die Verwendung von Pig Latin ein geeigneter und effizienter Ansatz zur Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen mit Hadoop ist. Dabei konnte der vermutete Zusammenhang zwischen der Ausführungszeit einer Anfrage und dem von der Anfrage erzeugten Datenaufkommen bestätigt werden. Besonders deutlich wurde dies durch die untersuchten Optimierungen, mit deren Hilfe die Ausführungszeit einer Anfrage teilweise deutlich reduziert werden konnte. Die verwendeten Datensatz-Größen von bis zu 1600 Millionen RDF-Tripeln übertreffen die Möglichkeiten von Systemen, die nur auf einem Computer ausgeführt werden, bereits um ein Vielfaches, was der Vergleich in [Sc09] belegt. Bedenkt man die Tatsache, dass für die Evaluation nur ein kleiner Hadoop-Cluster zum Einsatz kam (Yahoo! betreibt z.B. einen Hadoop-Cluster mit mehreren tausend Computern) und Pig sich in einer relativ frühen Entwicklungsphase befunden hat (die Evaluation wurde mit Pig 0.5.0 durchgeführt), wird das Potential des Ansatzes deutlich. Die vorgestellte Übersetzung bietet somit eine einfache und zugleich effiziente Möglichkeit, die Leistungsfähigkeit eines Hadoop-Clusters zur verteilten und parallelisierten Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen zu nutzen.

Literaturverzeichnis

- [Ab07] Abadi, D. J. et al.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proc. VLDB 2007; S. 411-422.
- [Ap10] Apache Software Foundation: Pig Latin Reference Manual. Hadoop Pig, 2010. <http://hadoop.apache.org/pig/docs/>
- [BHL01] Berners-Lee, T.; Hendler, J.; Lassila, O.: The Semantic Web. In: Scientific American, 2001.
- [BKH02] Broekstra, J.; Kampman, A.; Harmelen, F. van: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proc. ISWC 2002; S. 54-68.
- [Bo05] Boncz, P. et al.: Pathfinder: XQuery – The Relational Way. In: Proc. VLDB 2005; S. 1322-1325.
- [Ch09] Choi, H.; Son, J.; Cho, Y.; Sung, M.; Chung, D.: SPIDER: A System for Scalable, Parallel / Distributed Evaluation of large-scale RDF Data. In: Proc. CIKM 2009; S. 2087-2088.
- [Cy05] Cyganiak, R.: A relational algebra for SPARQL. TR. HP Laboratories Bristol. 2005.
- [DG04] Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Communications of the ACM, Vol. 51, Nr. 1, 2008; S. 107-113.

- [Er10] Erling, O.: Towards Web Scale RDF. In: SSWS, Karlsruhe, Germany, 2008.
- [Ga09] Gates, A. F. et al.: Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. In: VLDB Endow., Vol. 2, Nr.2, 2009; S. 1414-1425.
- [GPH05] Gui, Y.; Pan, Z.; Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. In: J. Web Sem., Vol. 3, Nr. 2-3, 2005; S. 158-182.
- [Ha07] Harth, A. et al.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Proc. ISWC/ASWC 2007, Vol. 4825; S. 211-224.
- [HBF09] Hartig, O.; Bizer, C.; Freytag, J.: Executing SPARQL Queries over the Web of Linked Data. In: Proc. ISWC 2009; S. 293-309.
- [HG03] Harris, S.; Gibbins, N.: 3store: Efficient Bulk RDF Storage. In: Proc. PSSS 2003; S. 1-20
- [HH07] Hartig, O.; Heese, R.: The SPARQL Query Graph Model for Query Optimization. In: ESWC 2007, Vol. 4519; S. 564-578.
- [HLS09] Harris, S.; Lamb, N.; Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: Proc. SSWS 2009; S. 94-109.
- [Hu10] Husain, M. F. et al.: Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. In: IEEE 3rd International Conference on Cloud Computing (CLOUD) 2010; S. 1-10.
- [Le10] DBLP Bibliography, 2010. <http://www.informatik.uni-trier.de/~ley/db/>
- [Mc01] McBride, B.: Jena: Implementing the RDF Model and Syntax Specification. In: Proc. of the 2nd International Workshop on the Semantic Web, 2001.
- [MMM04] Manola, F.; Miller, E.; McBride, B.: RDF Primer. World Wide Web Consortium (W3C), 2004. <http://www.w3.org/TR/rdf-primer/>
- [MT08] Mika, P.; Tummarello, G.: Web Semantics in the Clouds. In: IEEE Intelligent Systems, Vol. 23, Nr. 5, 2008; S. 82-87.
- [MYL10] Myung J.; Yeon J.; Lee S.: SPARQL Basic Graph Pattern Processing with Iterative MapReduce. In: Proc. MDAC 2010; S. 1-6.
- [NW08] Neumann, T.; Weikum, G.: RDF-3X: a RISC-style Engine for RDF. In: Proc. VLDB Endow., Vol. 1, Nr. 1, 2008; S. 647-659.
- [NW09] Neumann, T.; Weikum, G.: Scalable Join Processing on Very Large RDF Graphs. In: Proc. SIGMOD 2009; S. 627-640.
- [OI08] Olston, C. et al.: Pig Latin: A Not-So-Foreign Language for Data Processing. In: Proc. SIGMOD 2008; S. 1099-1110.
- [Or10] Oren, E. et al.: MARVIN: A platform for large-scale analysis of Semantic Web data. In: Proc. of the International Web Science Conference 2009.
- [Ow09] Owens, A. et al.: Clustered TDB: A Clustered Triple Store for Jena. In: WWW 2009.
- [PAG09] Pérez, J.; Arenas, M.; Gutierrez, C.: Semantics and Complexity of SPARQL. In: ACM Trans. Database Syst., Vol. 34, Nr. 3, 2009; S. 1-45.
- [PS08] Prud'hommeaux, E.; Seaborne, A.: SPARQL Query Language for RDF. World Wide Web Consortium (W3C), 2008. <http://www.w3.org/TR/rdf-sparql-query/>
- [QL08] Quilitz, B.; Leser, U.: Querying Distributed RDF Data Sources with SPARQL. In: Proc. ESWC 2008. LNCS, Vol. 5021; S. 524-538.
- [RDA10] Ravindra, P.; Deshpande, V.; Anyanwu, K.: Towards Scalable RDF Graph Analytics on MapReduce. In: Proc. MDAC 2010; S. 1-6.
- [RS10] Rohloff, K.; Schantz, R. E.: High Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: The SHARD Triple-Store.
- [Sc09] Schmidt, M. et al.: SP²Bench: A SPARQL Performance Benchmark. In: Proc. ICDE 2009; S. 222-233.
- [Sc10] Schätzle, A.: PigSPARQL: Eine Übersetzung von SPARQL nach Pig Latin, Masterarbeit. Lehrstuhl für Datenbanken und Informationssysteme, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, 2010.
- [St08] Stocker, M. et al.: SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In: Proc. WWW 2008; S. 595-604.

Koordinierte zyklische Kontext-Aktualisierungen in Datenströmen

Dennis Geesen¹, André Bolles¹, Marco Grawunder¹, Jonas Jacobi¹, Daniela Nicklas²,
H.-Jürgen Appelrath¹

¹Informationssysteme und ²Datenbank- und Internettechnologien
Department für Informatik
Universität Oldenburg
D-26121 Oldenburg

Abstract: Kontextsensitive Anwendungen benötigen ein möglichst exaktes Modell der Umgebung. Zur Ermittlung und regelmäßigen Aktualisierung dieses Kontextmodells werden typischerweise Sensordaten verwendet. Datenstrommanagementsysteme (DSMS) bilden die ideale Basis, um mit den durch die Sensoren generierten, potentiell unendlichen Datenströmen umzugehen. Leider bieten bisherige DSMS keine native Unterstützung für dynamische Kontextmodelle. Insbesondere die bei der Aktualisierung entstehenden Zyklen im Anfrageplan bedürfen einer besonderen Koordination, um Aktualität und Konsistenz des Kontextmodells zu gewährleisten. Diese Arbeit präsentiert eine Lösung, die einen Broker zur Koordination der verschiedenen Zugriffe auf das Kontextmodell als neuen Operator im DSMS einführt. Wir zeigen dazu eine semantische Beschreibung und eine abstrakte Implementierung des Brokers.

1 Einleitung

Zur effektiven Verwendung von Sensoren in kontextsensitiven Anwendungen ist die Verwaltung eines Kontextmodells notwendig. Im Bereich der Fahrerassistenzsysteme (FAS) beispielsweise spiegelt das Kontextmodell die derzeitige Umgebung des Fahrzeugs wider. Sensoren erzeugen dabei in einer hohen Rate eine große Menge an Daten, die zeitnah verarbeitet werden muss. Zur zeitnahen Verarbeitung solcher Daten können Datenstrommanagementsysteme (DSMS) eingesetzt werden. Diese adaptieren Anfrageverarbeitungsmechanismen, wie sie aus Datenbankmanagementsystemen (DBMS) bekannt sind und ermöglichen so eine flexible Verarbeitung potentiell unendlicher Datenströme [See04].

Bei der Umsetzung kontextsensitiver Anwendungen wie FAS muss jedoch das Kontextmodell verwaltet werden. Insbesondere die koordinierte Aktualisierung durch mehrere Sensoren spielt hier eine entscheidende Rolle. Hierbei entstehen mehrere Zyklen, indem das aktuelle Kontextmodell gelesen, durch einen Sensormesswert aktualisiert und dann wieder gespeichert wird. Diese zyklischen Kontext-Aktualisierungen müssen zeitlich korrekt ausgeführt werden, sodass das Kontextmodell fortlaufend und nicht beliebig aktualisiert wird und damit Aktualität und Konsistenz des Kontextmodells gewährleistet werden. Dabei ist jedoch nicht die zeitliche Reihenfolge des ersten Zugriffs eines Zyklus auf das Kontextmo-

dell wichtig, sondern die zeitliche Reihenfolge, in der die Sensormesswerte erfasst worden sind. Da Zyklen jedoch parallel verarbeitet werden und verschiedene Latenzen besitzen können, ist die Reihenfolge der Zugriffe auf das Kontextmodell nicht zwingend synchron mit der Reihenfolge der Messwerterfassung. Obwohl im ersten Ansatz DBMS und deren Transaktionskontrolle für die Koordinierung sinnvoll erscheinen, beruht die Transaktionskontrolle auf der zeitlichen Reihenfolge des ersten Zugriffs auf das Kontextmodells und nicht wie gewünscht auf die zeitliche Reihenfolge der Messwerterfassung, also der Daten selbst. Aus diesem Grund sind sowohl die – ohnehin nicht für kontinuierliche Daten adäquaten – klassischen DBMS als auch die schnelleren Hauptspeicher-DBMS für dieses Problem nicht geeignet.

Alle lesenden und schreibenden Zugriffe müssen anhand der Daten koordiniert werden. Insbesondere stellen zyklische Aktualisierungen eine besondere Herausforderung dar, da ein Zyklus komplett beendet sein muss, bevor ein neuer Zugriff gestattet werden darf. Die hochdynamische Verwaltung eines Kontextmodells in flexiblen DSMS wurde bisher nicht betrachtet. Aus diesem Grund stellen wir in dieser Arbeit einen Broker vor, der als Datenstrom-Operator das Speichern, sowie den koordinierten und zeitlich korrekten Zugriff auf das Kontextmodell erlaubt. Dazu stellt Kapitel 2 zunächst verwandte Arbeiten vor. Kapitel 3 führt das verwendete Modell und System des Datenstrommanagements ein. Im Anschluss werden in Kapitel 4 die Motivation und Anforderungen an eine Umsetzung erläutert. Darauf aufbauend beschreibt Kapitel 5 die Realisierung des Broker-Operators, indem zum einen ein logischer Operator für die Semantik und zum anderen ein physischer Operator für eine abstrakte Implementierung vorgestellt wird. Wir betrachten den Ansatz in Kapitel 6 in einem Experiment und geben in Kapitel 7 eine Übersicht und ein abschließendes Fazit.

2 Verwandte Arbeiten

Die Verarbeitung von Datenströmen auf Basis von Anfrageverarbeitungsmechanismen, die ähnlich zu denen in DBMS sind, wurde bisher unter anderem in Prototypen wie Aurora [ACc⁺03], Borealis [AAB⁺05], STREAM [ABB⁺03], PIPES [KS05] oder auch inzwischen in kommerziellen Produkten wie RTM [Rea10] oder SPADE [GAW⁺08] umgesetzt. Auch für die Verwaltung von Kontextmodellen in kontextsensitiven Anwendungen existieren Middlewares, wie z.B. Gaia [RHC⁺02], MAIS [CCMP06] oder Nexus [CEB⁺09]. Dabei integriert Nexus auch Datenstrommanagement (DSM)-Konzepte, auch wenn hier aufgrund der geringen Datenrate ein korrekter Zugriff auf das Kontextmodell auch dann gewährleistet werden kann, wenn dies in einem DBMS gespeichert wird. Das hier vorgestellte Konzept betrachtet jedoch eine hohe Datenrate und verwaltet das Kontextmodell daher direkt im DSMS.

Die hier als Motivation dienende Objektverfolgung wurde im Projekt STREAM erprobt, indem mehrere Datenstrom-Elemente den Bewegungsverlauf eines Objektes beschreiben [PS04]. Weitere Arbeiten beschäftigen sich ebenfalls mit kontextbezogenen Daten, wie beispielsweise [HJ04] mit ortsbasierten Anfragen oder [MA08] mit spatio-temporalen Daten. Hierbei wurden jedoch keine Kontextmodelle verwaltet. Die Betrachtung von Zyklen

in Datenströmen wurde unter anderem von [WRML08], [DS00] oder [GADI08] behandelt. Hier wurden aber lediglich rekursive Vereinigung, rekursiv verschachtelte XML-Fragmente oder Kleenesche Hüllen betrachtet, bei denen keine koordinierte Verarbeitung verschiedener Zyklen notwendig war. Des Weiteren führt [CGM09] einen neuen Operator ein, um zu erkennen wann eine transitive Hülle bei einer rekursiven Vereinigung abgeschlossen ist. Allerdings wird auch hier keine Koordinierung der zeitlichen Reihenfolge der Daten durchgeführt. [GBz06] beschäftigen sich mit Synchronisation und Konfliktserialisierbarkeit beim Einfügen neuer Elemente innerhalb eines Fensters, während Daten aus dem Fenster noch gelesen werden und zeigen hierzu eine entsprechende Scheduling-Strategie. Hierbei werden bereits geordnete lesende und geordnete schreibende Zugriffe betrachtet. In dieser Lösung werden jedoch zum einen unsortierte Zugriffe und zum anderen Zyklen berücksichtigt. Abschließend spielen hier auch Konzepte der Transaktionskontrolle in DBMS eine Rolle (s. [EN09] für eine Einführung).

3 Datenmodell und Anfrageverarbeitung

Ein Datenstrom ist eine potentiell unendliche Folge von Daten, die in der Regel als relationale Tupel repräsentiert werden. Ein Datenstrommanagementsystem (DSMS) erlaubt eine flexible und effiziente Verarbeitung solcher kontinuierlich auftretenden Daten, indem ähnlich zu Datenbankmanagementsystemen (DBMS) deklarative Anfragen verwendet werden. Zurzeit gibt es diverse Umsetzungen von DSMS, die meist in der Forschung als Prototyp eingesetzt werden oder aus einem solchen hervorgegangen sind (vgl. Abschnitt 2). Neben den genannten Systemen gibt es mit Odysseus [BGJ⁺09] ein Framework für DSMS, welches leicht erweiterbar und anpassbar ist. Dies erlaubt unter anderem eine einfache und flexible Evaluation neuer Forschungskonzepte. Die Architektur von Odysseus besteht, wie Abbildung 1 zeigt, aus mehreren Komponenten, die jeweils feste Funktionen (Fixpunkt) besitzen, die um sogenannte Variationspunkte erweitert werden [BGJ⁺09].

Die Anfrageverarbeitung kann in vier Phasen betrachtet werden. Zunächst sorgt die Übersetzungskomponente (Translate) dafür, dass eine deklarative Anfrage in einen Anfrageplan übersetzt wird. Hierzu bedient sich die Übersetzungskomponente einer logischen Algebra, die Semantik und Aufbau des Anfrageplans definiert. Der daraus entstandene logische Anfrageplan wird im nächsten Schritt durch die Restrukturisierungskomponente (Rewrite) unter Verwendung von Rewrite-Regeln optimiert. Dieser Plan wird in einem weiteren Schritt der Transformationskomponente (Transform) übergeben. Diese übersetzt den Plan auf Grundlage von Transform-Regeln und einer physischen Algebra in einen physischen Anfrageplan. Der physische Anfrageplan wird dann der Ausführung (Execute) übergeben und gegebenenfalls mit vorhandenen Anfrageplänen verbunden. Die linke Seite in Abbildung 1 zeigt einen gesamten Anfrageplan, wobei die Operatoren als *Pipe* dargestellt werden. Die zuvor genannten Variationspunkte sind beispielsweise durch die Möglichkeit gegeben, dass die Übersetzungskomponente um neue Sprachen erweitert werden kann. Des Weiteren können Übersetzung und Transformation um neue Algebra-Operatoren erweitert werden. Analog können auch die Restrukturierungs- und Transformationskomponente

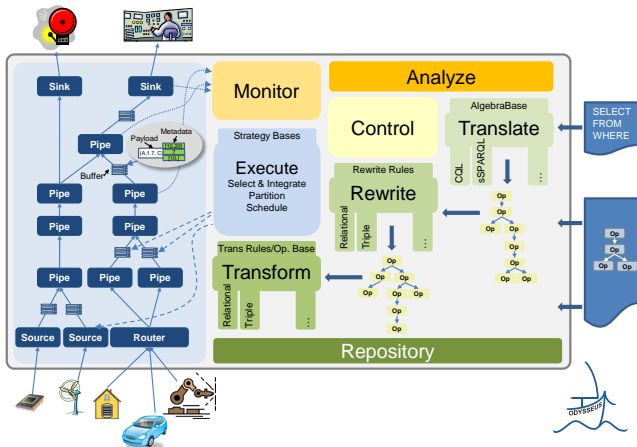


Abbildung 1: Architektur von Odysseus [BGJ⁺09]

um neue Regeln ergänzt werden. Außerdem lassen sich zum Beispiel neue Datenmodelle, Metadaten oder auch neue Scheduling-Strategien integrieren.

Der genannte Anfrageplan basiert auf einer Verknüpfung von Operatoren, die auf der logischen bzw. physischen Algebra basieren und daher entsprechend einen logischen bzw. physischen Anfrageplan darstellen. Hierbei dient die logische Algebra i. A. lediglich dazu, die genaue Semantik der einzelnen Operatoren festzulegen und auf Grundlage der erweiterten relationalen Algebra Optimierungen zu ermöglichen. Dabei betrachtet die logische Algebra alle Datenströme als temporale Multimengen und Operatoren stellen Abbildungen zwischen diesen Multimengen dar. Da hierbei die gesamte Menge bekannt sein muss, würden Operatoren entsprechend unendlich warten und blockieren. Daher hat die physische Algebra eine andere Sicht auf die Datenströme. Hier unterscheidet man zwischen nicht-blockierenden und blockierenden Operatoren. Nicht-blockierende Operatoren müssen ihre Eingabe nicht komplett konsumieren, um eine Ausgabe zu erzeugen. So kann z.B. eine Selektion direkt auf ein Element angewendet werden, indem es entweder weitergeleitet oder verworfen wird. Blockierende Operatoren hingegen sind meist zustandsbehaftete Operatoren, die zunächst die ganze Eingabe benötigen, um ein Ergebnis produzieren zu können. Beispielsweise kann eine Aggregation erst berechnet werden, wenn alle nötigen Elemente vorliegen. Um dieses blockierende Verhalten zu lösen, gibt es unter anderem den hier verwendeten Fenster-Ansatz, bei dem lediglich endliche Ausschnitte – in der Regel die aktuellsten Elemente – des Datenstroms betrachtet werden. Hierbei bekommt jedes Datenstromelement eine Gültigkeit, die entweder durch das Anheften eines Gültigkeitsintervall [KS05] oder durch Markierung mit einem positiven bzw. negativen Marker [GAE06] umgesetzt wird. Blockierungen können nun dadurch aufgelöst werden, dass in den einzelnen Operatoren nur noch Elemente gemeinsam betrachtet werden, die gleichzeitig gültig sind, sich also im selben Fenster befinden.

4 Motivation und Anforderungen

Kontextsensitive Anwendungen treffen ihre Entscheidung typischerweise auf Grundlage eines Kontextmodells, das die derzeitige Umgebung widerspiegelt. Dazu ist es notwendig, dass die aktuell gültige Instanz des Kontextmodells gespeichert und durch neue Messdaten fortlaufend aktualisiert und der Anwendung zu Verfügung gestellt wird. Hierzu zeigt Abbildung 2 eine beispielhafte Architektur, in der mit zwei Sensoren kontinuierlich Objekte in der Umgebung erfasst werden. Angenommen ein Fahrzeug sei mit einem Radarsensor

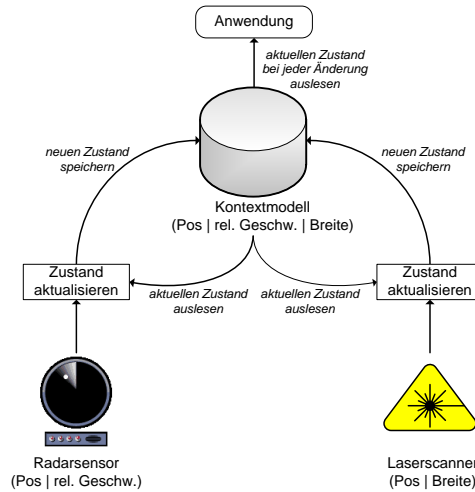


Abbildung 2: Beispielarchitektur

und einem Laserscanner ausgestattet. Der Radarsensor liefert die Position sowie die relative Geschwindigkeit detektierter Objekte. Der Laserscanner dagegen liefert die Position sowie die Breite detektierter Objekte. Anhand dieser Daten wird ein Kontextmodell mit allen drei Eigenschaften, Position, relative Geschwindigkeit und Breite aufgebaut. Wenn neue Objektdetektionen im System eintreffen, wird das aktuelle Kontextmodell aus dem Speicher gelesen. Daraufhin wird das Kontextmodell anhand der neuen Objektdetektionen im entsprechenden Teil des Anfrageplans (links Radar oder rechts Laserscanner) aktualisiert und anschließend wieder zurück in den Speicher geschrieben. Dabei ist die zeitliche Reihenfolge der Aktualisierungen an der zeitlichen Reihenfolge der erfassten Daten auszurichten. Werden bspw. Objekte vom Radarsensor zum Zeitpunkt t_1 und vom Laserscanner zum Zeitpunkt t_2 mit $t_1 < t_2$ erfasst, so ist das Kontextmodell zunächst mit den Objekten aus dem Radarsensor und anschließend mit denen aus dem Laserscanner zu aktualisieren. Andernfalls können Informationen verloren gehen oder bspw. in Vorhersagefunktionen falsch berechnet werden. Ist es in einer Anwendung möglich, dass Objekte von mehreren Sensoren gleichzeitig erkannt werden, so muss das Kontextmodell in diesem Fall durch einen speziellen Aktualisierungs-Operator, der mehrere Messwerte gleichzeitig verarbeiten kann, in einem eigenen Zyklus aktualisiert werden. Bei unsynchronisierten Sensoren tritt dieser Fall jedoch i. d. R. nicht auf, so dass er im Folgenden nicht näher betrachtet

wird. Da Sensoren nicht zwangsläufig synchronisiert sind, kann über die Reihenfolge des Erfassens der Daten im Vorfeld keine Aussage getroffen werden. Damit benötigt man für die zeitlich koordinierte Aktualisierung des Kontextmodells eine Art Transaktionskontrolle, die dynamisch anhand der eintreffenden Daten entscheiden kann, welcher Zyklus zur Aktualisierung des Kontextmodells genutzt wird. Die Transaktionskontrolle eines klassischen oder eines Hauptspeicher-DBMS kann hierzu, wie bereits erwähnt, nicht eingesetzt werden, da das entsprechende DBMS keine Kenntnis über die zeitliche Reihenfolge der Daten erhalten würde. Man benötigt also eine native Transaktionskontrolle für das Kontextmodell innerhalb des eingesetzten DSMS. Das in Abbildung 2 gezeigte Beispiel kann dabei als Anfrageplan in einem DSMS umgesetzt werden, wie Abbildung 3 zeigt. Dabei

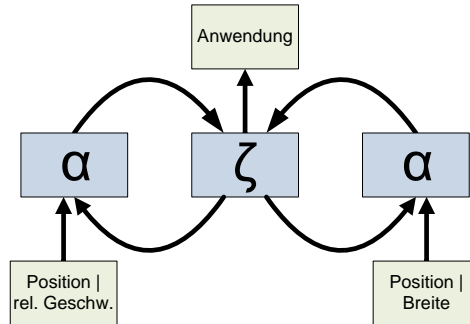


Abbildung 3: Umsetzung des Beispiels als Anfrageplan

wird das Kontextmodell durch einen Broker-Operator (ζ) verwaltet und jeweils durch Aktualisierungs-Operatoren (α) aktualisiert. Die Aktualisierungen werden dabei nicht von dem Broker selbst übernommen sondern in zusätzlichen Aktualisierungs-Operatoren ausgelagert, da für jeden Sensor gegebenenfalls verschiedene Aktualisierungs-Algorithmen von einfachen Berechnungen bis zur Verwendung von Prädiktionsfunktionen existieren. Die Verwendung von Aktualisierungs-Operatoren schafft somit eine höhere Flexibilität und erlaubt es u.a. auch, dass zwischen Broker-Operator und Aktualisierungs-Operator ggf. weitere Operatoren existieren, die das übertragene Kontextmodell u.U. modifizieren. Aus diesen Gründen sei an dieser Stelle von der konkreten Funktionalität von α abstrahiert, um lediglich auf die genaue Funktionsweise des Brokers einzugehen.

Anforderungen und Konzepte

Wird die Verwaltung eines Kontextmodells wie oben beschrieben durch ein DSMS umgesetzt, so ergeben sich dadurch Anforderungen an den Broker. Zum einen muss der Broker Mechanismen bereitstellen, die einen lesenden und schreibenden Zugriff auf das Kontextmodell erlauben. Zum anderen muss der Broker Eigenschaften eines DSMS, insbesondere die zeitliche Ordnung und die push-basierte Verarbeitung, berücksichtigen. Im Folgenden werden daher die Anforderungen und die jeweiligen Konzepte anhand der verschiedenen Zugriffsmöglichkeiten beschrieben und entsprechende Konzepte vorgestellt, die eine Inte-

gration in ein DSMS erlauben.

Schreibender Zugriff Ein schreibender Zugriff ist über eingehende Datenströme, den sogenannten *Update Streams*, mit dem Broker Operator möglich. Ein Update Stream liefert Datenelemente, die im Kontextmodell gespeichert werden sollen. Solche Elemente können zum einen Aktualisierungen des Kontextmodells zum anderen aber auch neue Elemente sein, die noch nicht im Kontextmodell vorhanden sind. Beim Eintreffen neuer Daten über einen Update Stream wird der aktuelle Zeitfortschritt festgestellt und alte, nicht mehr gültige Daten aus dem Kontextmodell entfernt. Neue Elemente werden einfach hinzugefügt. Aktualisierungen ersetzen jeweils ihre Vorgängerversion.

Lesender Zugriff Der lesende Zugriff auf den Broker kann in zwei unterschiedliche Zugriffsarten unterteilt werden. Zum einen gibt es den kontinuierlich lesenden Zugriff, der über sogenannte *Observation Streams* realisiert wird. Jede Änderung des Kontextmodells wird über diese Observation Streams bspw. an nachfolgende Anwendungen weitergeleitet. Zum anderen gibt es aber auch den einmalig lesenden Zugriff, der bei Vorliegen neuer Messwerte am Aktualisierungs-Operator das jeweils aktuelle Kontextmodell zurückliefern soll. Da dieses Kontextmodell jedoch am Aktualisierungs-Operator vorliegt, kann der Broker nicht wissen, wann das aktuelle Kontextmodell ausgeliefert werden soll. Daher werden für diese Art des Zugriffs zwei Arten von Datenströmen eingeführt. Über sogenannte *Request Streams* kann bei Vorliegen neuer Messwerte aus den Sensoren das jeweils aktuelle Kontextmodell beim Broker angefragt werden. Dieses wird dann über sogenannte *Response Streams* an den Aktualisierungs-Operator weitergeleitet.

Für einen Aktualisierungszyklus erhält man damit die in Abbildung 4 gezeigten Datenströme. Trifft hier ein neuer Messwert bei dem Aktualisierungs-Operator ein, so kann die-

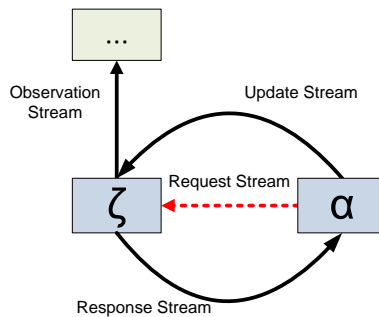


Abbildung 4: Stromtypen

ser einen Request an den Broker schicken, indem er ein entsprechendes Element über den Request Stream übergibt. Dann kann der Broker diesen Request bearbeiten und schickt entsprechend das aktuelle Kontextmodell über den zugehörigen Response Stream an den Aktualisierungs-Operator. Dieser kann nun das Kontextmodell anhand des Messwertes

und des aktuellen Kontextmodells aktualisieren. Danach wird das Kontextmodell zurück an den Broker übergeben. Da sich hier nun noch andere, gegebenenfalls nicht mehr gültige, Elemente im Broker befinden, wird das gespeicherte Kontextmodell im Broker zunächst anhand der Aktualisierung bereinigt. Danach wird dann die Aktualisierung dem Kontextmodell hinzugefügt. Der Broker beinhaltet danach entsprechend wieder das gültige Kontextmodell. Erst jetzt kann ggf. ein anderer Zyklus vom Broker bedient werden.

Ein hier verwendeter Request über den Request Stream besteht nur aus einem Zeitstempel, da nur der Zeitpunkt des zugehörigen Messwertes im Aktualisierungs-Operator und nicht die Daten des Requests selbst für eine zeitliche Sortierung der Requests notwendig sind. Hierbei wäre es ebenso denkbar, dass neben dem Zeitstempel zusätzliche Daten übertragen werden, um bspw. nur einen Teil des Kontextmodells auszuwählen, der vom Broker als Antwort an den Aktualisierungs-Operator geschickt wird. Da dies jedoch stark von der Anwendung und den genutzten Aktualisierungs-Algorithmen abhängt und hier davon abstrahiert wird, wird das Kontextmodell als ein Gesamtes betrachtet.

5 Realisierung

Die Realisierung des Broker-Operators, der die im vorherigen Abschnitt genannten Anforderungen und Konzepte umsetzt, wird in zwei Ebenen betrachtet. Die Umsetzung als logischer Operator erlaubt eine semantisch korrekte Formulierung des Brokers und erlaubt dadurch u.a. Optimierungen auf Grundlage einer Algebra. Das logische Konzept kennt jedoch alle Daten im Voraus, was jedoch nicht effizient implementiert werden könnte, so dass der physische Broker-Operator eine abstrakte Implementierung auf Grundlage eines erweiterten Konzeptes darstellt.

5.1 Logischer Operator

Zur Definition der Semantik des Brokers wird ein logischer Broker-Operator auf Basis der logischen Algebra aus [Krä07] formal beschrieben. Sei dazu \mathbb{S}^l die Menge aller logischen Datenströme. Dann ist ein Datenstrom $S_\tau^l \in \mathbb{S}^l$ als eine Multimenge von Elementen (e, t, n) definiert, wobei $e \in \Omega_\tau$ ein Tupel vom Typ τ ist. Des Weiteren ist $n \in \mathbb{N}$ mit $n > 0$ die Anzahl des Tupels e zum Zeitpunkt $t \in T$. Hierbei ist T die Menge aller Zeitstempel in einem diskreten und total geordneten Wertebereich $\mathbb{T} = (T, \leq)$. Somit gibt beispielsweise ein Element $(a, 4, 3)$ an, dass das Tupel a zum Zeitpunkt 4 dreimal vorkommt.

Sei der Broker $\zeta : (\mathbb{S}_\tau^l)^k \times \mathbb{F}_{next} \rightarrow (\mathbb{S}_\tau^l)^m$ mit $k, m \in \mathbb{N}, k, m \geq 1$ als eine Abbildung von k logischen Datenströmen und einer Auswahlfunktion auf m logische Datenströme definiert. Entsprechend wird ein Update Stream oder ein Request Stream mit $S_{in}^l \in (\mathbb{S}_\tau^l)^k$ und analog ein Observation oder Response Stream mit $S_{out}^l \in (\mathbb{S}_\tau^l)^m$ bezeichnet. Des Weiteren beschreibt eine Auswahlfunktion $f_{next} \in \mathbb{F}_{next}$ mit $f_{next} : T \rightarrow \mathcal{P}(S_{out}^l)$ eine Abbildung von einem Zeitstempel auf eine Menge von ausgehenden Datenströmen. Sie entspricht der Umsetzung der Request Streams, indem sie zu einem Zeitpunkt t die Menge

von Response Streams oder Observation Streams liefert, die zu diesem Zeitpunkt bedient werden müssen. Hierzu sei $S_{req}^l \subseteq S_{in}^l$ die Menge der Request Streams aus allen eingehenden Datenströmen. Dann ist $R : S_{req}^l \rightarrow S_{out}^l$ eine Abbildung, die jedem eingehenden Request Stream einen Ausgangsdatenstrom zuordnet. Dann wäre

$$f_{next}(t) := \{\hat{S} \in S_{out}^l | \forall S \in R^{-1}(\hat{S}) : \exists (e, t, n) \in S\} \quad (1)$$

eine Auswahlfunktion, die eine Menge von Response Streams \hat{S} liefert, zu denen im zugehörigen Request Stream $R^{-1}(\hat{S})$ ein Tupel (e, t, n) und damit ein Request vorliegt.

Seien S_1 bis S_k logische eingehende Datenströme und \hat{S}_1 bis \hat{S}_m logische ausgehende Datenströme. Dann ist der Broker definiert durch

$$\begin{aligned} \zeta_{f_{next}}(S_1, \dots, S_k) &:= (\hat{S}_1, \dots, \hat{S}_m) \text{ mit} \\ \hat{S}_j &:= \{(e, \hat{t}, \hat{n}) | \exists X \subseteq S. \\ &\quad X \neq \emptyset \quad \wedge \\ &\quad X = \{(e, t, n) \in S | n(t) = n(\hat{t})\} \quad \wedge \\ &\quad \hat{n} = \sum_{(e, t, n) \in X} n \quad \wedge \\ &\quad S := \{(e, \hat{t}, \bar{n}) | ((e, \hat{t}, n_1) \in S_1 \wedge S_1 \notin S_{req}^l \vee n_1 = 0) \wedge \dots \wedge \\ &\quad ((e, \hat{t}, n_k) \in S_k \wedge S_k \notin S_{req}^l \vee n_k = 0) \quad \wedge \\ &\quad \bar{n} > 0 \quad \wedge \\ &\quad \bar{n} = \sum_{i=1}^k n_i\} \quad \wedge \\ &\quad \hat{S}_j \in f_{next}(\hat{t})\} \end{aligned} \quad (2)$$

wobei $n(t)$ die Anzahl der Elemente des Datenstromes bis zum Zeitpunkt t ist, welche durch

$$n(t) = |\{(e, \tilde{t}, n) \in S | \tilde{t} \leq t\}| \quad (3)$$

definiert ist.

Zu einem Zeitpunkt \hat{t} werden alle eingehenden Datenströme S_1, \dots, S_k zu einem Datenstrom S zusammengefasst, sofern es kein Tupel $(S_i, \tilde{S}) \in R$ gibt, sodass es sich bei S_i nicht um einen Request Stream handelt. \bar{n} gibt an, dass gleiche Tupel in unterschiedlichen Eingangsströmen aufsummiert werden. Aus diesem zusammengeführten Datenstrom S nimmt der Broker das Element (e, t, n) , das mit $n(t)$ dieselbe Anzahl an Vorgängern hat, wie es zum aktuellen Zeitpunkt mit $n(\hat{t})$ gibt. Hiermit wird zeitlich betrachtet das letzte Element genommen, das vor \hat{t} gültig ist. Dadurch erhält man in Bezug auf das Kontextmodell den letzten gültigen Zustand zum Zeitpunkt \hat{t} . Das so ausgewählte Tupel wird an alle ausgehenden Datenströme \hat{S}_j übergeben, die von der Auswahlfunktion f_{next} bestimmt worden sind. Somit bekommen zum Zeitpunkt \hat{t} alle Datenströme \hat{S}_j die Tupel, die zuletzt gültig waren.

Beispiel

Im Folgenden hat der Broker drei eingehende Datenströme S_1, S_2, S_3 und S_4 , sowie drei ausgehende Datenströme \hat{S}_1, \hat{S}_2 und \hat{S}_3 . Seien die eingehenden Datenströme wie folgt definiert:

$$\begin{aligned} S_1 &:= \{(c, 1, 1), (a, 2, 3), (a, 3, 3), (b, 3, 1), (a, 4, 3), (b, 4, 1), (c, 4, 1), (b, 5, 2), (b, 6, 2)\} \\ S_2 &:= \{(c, 1, 1), (a, 2, 1), (a, 5, 1), (b, 6, 1)\} \\ S_3 &:= \{(b, 2, 2), (b, 3, 2), (a, 4, 1), (b, 4, 1), (c, 4, 1), (a, 5, 2), (b, 5, 1), (a, 6, 1), (c, 6, 2)\} \\ S_4 &:= \{(b, 3, 1), (b, 6, 1)\} \end{aligned}$$

Dann zeigt Tabelle 1 den jeweiligen Zustand der vier Eingangsdatenströme S_1, S_2, S_3 und S_4 zu einem Zeitpunkt t .

t	S_1	S_2	S_3	S_4
1	$\langle c \rangle$	$\langle c \rangle$	$\langle \rangle$	$\langle \rangle$
2	$\langle a, a, a \rangle$	$\langle a \rangle$	$\langle b, b \rangle$	$\langle \rangle$
3	$\langle a, a, a, b \rangle$	$\langle \rangle$	$\langle b, b \rangle$	$\langle b \rangle$
4	$\langle a, a, a, b, c \rangle$	$\langle \rangle$	$\langle a, b, c \rangle$	$\langle \rangle$
5	$\langle b, b \rangle$	$\langle a \rangle$	$\langle a, a, b \rangle$	$\langle \rangle$
6	$\langle b, b \rangle$	$\langle b \rangle$	$\langle a, c, c \rangle$	$\langle b \rangle$

Tabelle 1: Beispiele für vier logische Datenströme

Seien weiter S_2 und S_4 Request Streams für die Ausgangsdatenströme \hat{S}_1 und \hat{S}_2 , sodass nachfolgende Zuordnungen existieren: $R(S_2) = \hat{S}_1$ und $R(S_4) = \hat{S}_2$.

Dann kann unter Verwendung der Auswahlfunktion f_{next} der Broker-Operator auf die Eingangsdatenströme S_1, S_2, S_3 und S_4 und entsprechend auf die drei Ausgangsdatenströme \hat{S}_1, \hat{S}_2 und \hat{S}_3 angewendet werden. Tabelle 2 zeigt entsprechend zu den Eingaben das Ergebnis von $\zeta_{f_{next}}(S_1, S_2, S_3, S_4)$.

t	\hat{S}_1	\hat{S}_2	\hat{S}_3
1	$\langle c \rangle$	$\langle \rangle$	$\langle c \rangle$
2	$\langle a, a, a, b, b \rangle$	$\langle \rangle$	$\langle a, a, a, b, b \rangle$
3	$\langle \rangle$	$\langle a, a, a, b, b, b \rangle$	$\langle a, a, a, b, b, b \rangle$
4	$\langle \rangle$	$\langle \rangle$	$\langle a, a, a, a, b, b, c, c \rangle$
5	$\langle a, a, b, b, b \rangle$	$\langle \rangle$	$\langle a, a, b, b, b \rangle$
6	$\langle a, b, b, c, c \rangle$	$\langle a, b, b, c, c \rangle$	$\langle a, b, b, c, c \rangle$

Tabelle 2: Broker-Operation über die logischen Datenströme S_1, S_2, S_3, S_4

Wie zu sehen ist, vereinigt der Broker alle Elemente, die nicht aus einem Request Stream kommen. Zum Zeitpunkt $t = 2$ werden daher nur die Elemente $\langle a, a, a \rangle$ aus dem Datenstrom S_1 und die Elemente $\langle b, b \rangle$ aus dem Datenstrom S_3 zusammengeführt. Da zu diesem Zeitpunkt eine Anforderung $\langle a \rangle$ von dem Request Stream S_2 vorliegt,

muss das Ergebnis $\langle a, a, a, b, b \rangle$ auch an den Ausgangsdatenstrom $R(S_2) = \hat{S}_1$ geliefert werden. Daher ist die Auswahlfunktion zu diesem Zeitpunkt wie folgt definiert: $f_{next}(2) = \{\hat{S}_1, \hat{S}_3\}$. Hierbei kann man unter anderem erkennen, dass die Nutzdaten von Elementen aus Request Streams ignoriert werden, da lediglich die Zeitstempel von Interesse sind. Wie bereits in Abschnitt 4 genannt, wird dann entsprechend auch das ganze Kontextmodell wiedergegeben, indem keine Auswahl für einen Teil des Kontextmodells als Nutzdaten mitgegeben wird.

5.2 Physische Algebra

Eine direkte Implementierung des logischen Algebra-Operators ist nicht praktikabel, da sonst für jeden Zeitpunkt ein eigenes Element im Datenstrom verarbeitet werden müsste. Die physische Algebra (vgl. [Krä07]) betrachtet daher eine kompaktere Sichtweise auf Datenströme. Sei dazu \mathbb{S}^p die Menge aller physischen Datenströme und $S_\tau^p \in \mathbb{S}^p$ ein physischer Datenstrom. Dann ist S_τ^p eine Menge von Elementen $(e, [t_s, t_e])$, wobei $e \in \Omega_\tau$ ein Element vom Typ τ ist und $[t_s, t_e)$ ein rechts-halboffenes Zeitintervall mit $t_s, t_e \in T$ ist. Hierbei ist t_s der Startzeitstempel und gibt den Zeitpunkt eines atomar auftretenden Messwertes an. Ferner ist t_e der Endzeitstempel, der das Gültigkeitsende eines Messwertes angibt, welches i.d.R. durch ein Fenster zugewiesen wurde. Hierbei ist T die Menge aller Zeitstempel mit $\mathbb{T} = (T, \leq)$.

5.2.1 Kontextmodell-Speicher

Der Kontextmodell-Speicher stellt im Prinzip nur einen Ausschnitt der schreibenden Datenströme dar. Da ein ähnliches Verhalten auch bei anderen Operatoren zu finden ist, wurde durch [DSTW02] eine *SweepArea* definiert, die es erlaubt Ausschnitte eines Datenstroms effizient zu speichern, zu durchsuchen und zu aktualisieren. Die *SweepArea* bietet dazu unter anderem die Methoden `insert` zum Hinzufügen, `iterator` zum sortierten Durchlaufen (bzgl. \leq_{t_s}) und `purgeElements` zum Bereinigen an. `purgeElements` bereinigt den Inhalt der *SweepArea* anhand eines Prädikats p_{remove} , indem die Methode nur solche Elemente entfernt, die sich über das Prädikat qualifiziert haben. Damit die *SweepArea* zum Löschen und zum Aktualisieren verwendet werden kann, wird folgendes Prädikat verwendet:

$$p_{remove}^\zeta(s, \hat{s}) := \begin{cases} true & \text{wenn } (t_s \geq \hat{t}_s \wedge p_{equal}(s, \hat{s})) \vee t_s \geq \hat{t}_e \\ false & \text{sonst} \end{cases}$$

Jedes neue Element s wird mit jedem Element \hat{s} aus der *SweepArea* geprüft, ob sie zusammen das Prädikat erfüllen. Dabei wird geprüft, ob es sich bei s um eine neuere Version handelt. Das ist der Fall, wenn es einen größeren Startzeitstempel als \hat{s} besitzt und sie gemeinsam das Gleichheitsprädikat p_{equal} erfüllen. Ist das nicht der Fall, wird geprüft, ob das Element \hat{s} nicht mehr gültig ist. Dies ist der Fall wenn der Startzeitstempel von s größer als der Endzeitstempel von \hat{s} ist. Da das Gleichheitsprädikat p_{equal} abhängig vom Kontextmodell ist, wird es hier nicht explizit aufgeführt.

5.2.2 Auswahlfunktion

Um die Auswahlfunktion umzusetzen, wie sie im vorigen Abschnitt als f_{next} beschrieben ist, wird zunächst die Abbildung R benötigt. Hierzu sei ein abstrakter Datentyp (ADT) Metadatenverzeichnis definiert, der folgende Methoden bereitstellt:

getObservationStreams() Liefert alle Observation Streams zurück.

getResponseStream(Datenstrom S) Liefert den Response Stream $\hat{S} \in S_{out}^l$ zu dem Request Stream $S \in S_{req}^l$.

isRequestStream(Datenstrom S) Liefert wahr zurück, wenn es sich bei S um einen Request Stream handelt.

getRequests() Liefert eine Prioritätsqueue bzgl. eines Zeitstempels, die alle Request Streams enthält, zu denen noch Anfragen ausstehen.

Die Methode `getResponseStream` setzt dabei die Abbildung R und `isRequestStream` den Term $S \in S_{req}^l$ um. Ferner liefert `getRequests` eine Prioritätsqueue, die alle Requests liefert. Hierbei sind alle Requests aus den Request Streams anhand ihres Startzeitstempels einsortiert. Durch die Prioritätsqueue wird unter anderem die in Definition 2 verwendete Bedingung $n(t) = n(\hat{t})$ umgesetzt, indem nur das letzte Element ausgegeben wird, welches zum aktuellen Zeitpunkt t an der Reihe ist. Des Weiteren beinhaltet `getRequests` bereits nur noch Request Streams, so dass statt der Abbildung R^{-1} direkt die Abbildung R verwendet werden kann. Unter Verwendung dieses Metadatenverzeichnisses kann die Auswahlfunktion wie folgt umgesetzt werden:

Algorithm 1 getNext(Metadatenverzeichnis M , Zeitstempel t)

Require: Metadatenverzeichnis M , Zeitstempel t

Ensure: Eine Menge O aus physischen Datenströmen S^p

```

1:  $O \leftarrow M.getObservationStreams()$ 
2: loop
3:    $r := (\hat{S}, \hat{t}) \leftarrow M.getRequests().peek()$ 
4:   if  $\hat{t} \leq t$  then
5:      $M.getRequests().poll()$ 
6:      $S = M.getResponseStream(\hat{S})$ 
7:      $O.insert(S)$ 
8:   else
9:     break
10:  end if
11: end loop
12: return  $O$ 

```

Algorithmus 1 zeigt eine abstrakte Implementierung der Auswahlfunktion. Hierzu benötigt die Methode das genannte Metadatenverzeichnis, sowie den Parameter t , wie er auch in Definition 1 angegeben ist.

5.2.3 Physischer Operator

Um die Semantik des Brokers umzusetzen, wie sie in Definition (2) als logischer Broker-Operator beschrieben ist, muss bei der physischen Umsetzung noch der zeitliche Verlauf berücksichtigt werden. Ein physischer Operator konsumiert alle Elemente nur nacheinander. Daraus resultiert, dass auch der Broker zu einem Zeitpunkt t nicht wissen kann, ob noch Elemente folgen, dessen Zeitstempel kleiner als t sind. Im Falle der Request-Datenströme bedeutet dies, dass der Broker erst eine Anforderung bedienen darf, wenn er sicher gehen kann, dass nicht noch eine Anforderung eintrifft, die davor liegt. Um dieses blockierende Verhalten aufzulösen, bedient man sich einer Grundvoraussetzung von Datenströmen, bei der alle Elemente eines Datenstroms zeitlich anhand des Startzeitstempels sortiert sind, da dieser den eigentlichen atomaren Zeitpunkt eines Datenstromelements festlegt. Kommt demnach ein Element $(e, [t_s, t_e])$ am Operator an, so kann dieser davon ausgehen, dass aus demselben Eingang nur noch Elemente folgen werden, deren Startzeitstempel größer oder gleich t_s sind. Für ein Element muss der Broker also auf alle Eingänge entsprechend warten, bis er sichergehen kann, dass keine jüngeren Elemente mehr ankommen. Hierzu verwendet der Broker einen Zeitstempel \min_{t_s} , bei dem der Broker sichergehen kann, dass aus keinem Eingang mehr Elemente kleiner als \min_{t_s} folgen. Dieses Minimum kann dadurch gebildet werden, dass der Broker sich zu jedem Eingang i merkt, welchen Startzeitstempel t_{in_i} das letzte Element hatte. Das Minimum aller t_{in_i} ergibt entsprechend das globale Minimum, sodass $\min_{t_s} := \min(t_{in_1}, \dots, t_{in_k})$, wobei k der Anzahl der Eingangsdatenströme entspricht. Durch Berücksichtigung dieses Minimums, der zuvor beschriebenen *SweepArea* und der Auswahlfunktion `getNext` kann der Broker, wie im folgenden Algorithmus umgesetzt werden:

Algorithm 2 Broker-Operator

Require: Physische Datenströme $S_{in_1}, \dots, S_{in_k}$

Ensure: Physische Datenströme $S_{out_1}, \dots, S_{out_m}$

- 1: Sei M ein Metadatenverzeichnis mit den Datenströmen S_{in} und S_{out} , sowie einer Zuordnung von Request-Datenströmen zu Ausgangsdatenströmen
- 2: $t_{in_1}, \dots, t_{in_k}, \min_{t_s} \in T \cup \{\perp\}; t_{in_i} \leftarrow \perp$ mit $1 \leq i \leq k; \min_{t_s} \leftarrow \perp$
- 3: Sei SA eine leere *SweepArea*($\leq_{t_s}, p_{remove}^\zeta$)
- 4: Sei buf eine Prioritätsqueue für Elemente $(e, [t_s, t_e])$ mit Ordnungsrelation \leq_{t_s}
- 5: **for** $s := (e, [t_s, t_e]) \leftarrow S_{in_j}$ **do**
- 6: $t_{in_j} \leftarrow t_s$
- 7: $\min_{t_s} \leftarrow \min(t_{in_1}, \dots, t_{in_k})$
- 8: **if** $M.isRequestStream(S_{in_j})$ **then**
- 9: $M.getRequests().offer((S_{in_j}, t_s))$
- 10: **else**
- 11: $buf.offer(s)$
- 12: **end if**
- 13: **if** \min_{t_s} **not** \perp **then**
- 14: **while not** $buf.isEmpty()$ **do**
- 15: $\hat{s} := (\hat{e}, [\hat{t}_s, \hat{t}_e]) \leftarrow buf.peek()$
- 16: **if** $\hat{t}_s \leq \min_{t_s}$ **then**
- 17: $\hat{s} \leftarrow buf.poll()$

```

18:      SA.purgeElements( $\hat{s}$ )
19:      SA.insert( $\hat{s}$ )
20:    else
21:      break
22:    end if
23:  end while
24:  Iterator it = getNext(M,  $\min_{t_s}$ )
25:  while not it.hasNext() do
26:     $n := (S_o, t_o) \leftarrow it.next()$ 
27:    for all  $\tilde{s} \leftarrow SA.iterator()$  do
28:       $\tilde{s} \hookrightarrow S_o$ 
29:    end for
30:  end while
31: end if
32: end for

```

Der Algorithmus wird für jedes ankommende Element s ausgeführt. Dabei wird zunächst das Minimum für den zugehörigen Eingangsdatenstrom gesetzt und danach das globale Minimum berechnet (Zeile 6–7). Darauf folgend wird geprüft, ob es sich um einen Request Stream handelt. Wenn dem so ist, wird dem Metadatenverzeichnis eine neue Anforderung übergeben. Falls nicht, wird das ankommende Element zunächst in einem Puffer abgelegt (Zeile 8–12). Ist in einem weiteren Schritt dann $\min_{t_s} \neq \perp$, sodass mindestens aus jedem Eingangsdatenstrom ein Zeitstempel t_{in} vorliegt, dann wird zunächst der Puffer behandelt. Hierzu werden nacheinander die Elemente \hat{s} aus dem Puffer geholt. Diese werden dann mit *purgeElements* benutzt, um die *SweepArea* *SA* zu bereinigen. Hierbei werden, wie oben beschrieben, alle Elemente aus der *SweepArea* entfernt, die laut p_{remove}^ζ nicht mehr gültig sind. Anschließend wird das Element \hat{s} der *SweepArea* hinzugefügt. Der Inhalt der *SweepArea* entspricht somit dem aktuellen Zustand des Kontextmodells (Zeile 13–18). Ist jedoch ein Element \hat{s} zeitlich vor \min_{t_s} , dann darf es noch nicht aus dem Puffer geholt werden, da eventuell erst andere Eingaben und Requests abgearbeitet werden müssen (Zeile 16 bzw. 21). Somit dient dies auch der zeitlich korrekten Aktualisierung des Kontextmodells. Wenn alle aktuell gültigen Daten aus dem Puffer geholt wurden, dann werden mit *getNext* alle Ausgangsdatenströme bestimmt, die zum Zeitpunkt \min_{t_s} ausgeführt werden müssen. Anschließend wird der gesamte Inhalt der *SweepArea*, also das gesamte aktuell gültige Kontextmodell, an die vorher ausgewählten Ausgangsdatenströme geschickt (Zeile 24–30).

5.2.4 Optimierungen

Aus der relationalen Algebra sind Optimierungsregeln bekannt, die es erlauben einen Anfrageplan für eine effizientere Ausführung umzustellen, ohne dabei die Ergebnisse zu verändern. Ähnliche Regeln existieren auch für die relationale Algebra auf Datenströmen. Bei diesen Optimierungsregeln werden jedoch nur azyklische Anfragepläne betrachtet. Mit der Einführung des Broker-Operators werden jedoch Zyklen in Anfragepläne integriert, die sich in gewissen Anwendungsszenarien nicht vermeiden lassen. Dennoch lassen

sich auch hier Teile entsprechender Anfragepläne unter gewissen Umständen optimieren. Hierzu muss ein Anfrageplan mit Zyklen jeweils am Broker geteilt werden. Die dabei entstehenden Teilpläne sind azyklisch und lassen sich mit den gleichen Optimierungsregeln umstrukturieren, wie es auch in [Krä07] der Fall ist. Selbst eine Optimierung über den Broker hinweg ist möglich. So kann bspw. eine Selektion in einem Teilplan, der einen Observation Stream darstellt, vor den Broker gesetzt werden, wenn die Selektion nur Objekte herausfiltert, die nicht zur Aktualisierung anderer Objekte des Kontextmodells benötigt werden. Ein Beispiel könnte wie folgt sein. Sensoren am Fahrzeug detektieren auf einer Autobahn alle Objekte vor dem eigenen Fahrzeug inklusive der Fahrspur, auf der sich die Objekte befinden. Wenn eine Anwendung jedoch nur die Objekte auf der eigenen Fahrspur benötigt, würde sie einen Observation Stream, also eine Anfrage am Broker registrieren, in der eine Selektion auf die entsprechende Fahrspur enthalten ist. Wenn keine anderen Anwendungen existieren, die auch Fahrzeuge auf anderen Fahrspuren benötigen, dann können die Fahrzeuge, die sich nicht auf der eigenen Fahrspur befinden, bereits herausgefiltert werden, bevor sie im Kontextmodell abgespeichert werden. Damit reduziert sich dann der Aufwand für die Aktualisierung des Kontextmodells.

6 Experimente

Die Funktionsweise des Brokers wurde durch Experimente geprüft, da keine vergleichbaren Algorithmen bzw. Operatoren in DSMS für eine sinnvolle Evaluation existieren. Ferner erfolgte eine Implementierung in das Datenstrom Management Framework Odysseus [BGJ⁺09]. Hierbei wurde der im vorigen Abschnitt beschriebene physische Operator umgesetzt und integriert. Aufbauend auf diesem Operator wurde ein beispielhafter Anfrageplan in Anlehnung an die in Abschnitt 4 Motivation im System installiert, wie er in Abbildung 5 gezeigt wird. Dieser Plan beinhaltet zwei Datenquellen (A und B), die je-

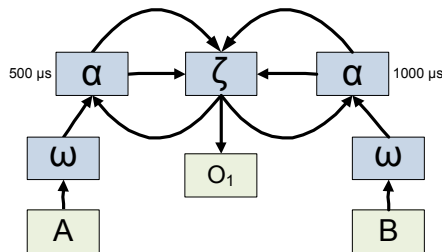


Abbildung 5: Anfrageplan der Experimente

weils einen Sensor simulieren sollen. Diese haben beide eine feste Frequenz und schicken immer abwechselnd, ebenfalls in einer festen Frequenz, neue Elemente in aufsteigender Reihenfolge bzgl. der Startzeitstempel an Odysseus. Die Elemente gelangen zunächst in einen Fenster-Operator ω , das jedem Element eine Gültigkeitsdauer zuweist, sodass Elemente später auch vom Broker entfernt werden können. Anschließend gelangen die Daten in die Aktualisierungs-Operatoren α . Diese fordern daraufhin mit einem Request die Daten

beim Broker an. Wenn der Broker auf Grundlage seiner Semantik den Request bedienen kann, liefert er das aktuell gültige Kontextmodell an den Aktualisierungs-Operator. Diese aktualisieren anhand des Elements und des übergebenen Kontextmodells den aktuellen Zustand. Um hierbei die Verarbeitungsdauer einer solchen Aktualisierung zu simulieren, wird auf der linken Seite $500\ \mu s$ und auf der rechten Seite $1000\ \mu s$ gewartet, bis das Ergebnis wieder dem Broker übergeben wird. Damit ist entsprechend ein Aktualisierungsschritt im Zyklus abgeschlossen. Ferner wird dazu die linke Aktualisierung als Zyklus A und die rechte als Zyklus B bezeichnet. Dies bedeutet, dass sowohl ein Zyklus als auch ein Sensor feste Frequenzen haben. Aus diesem Grund wurde in der Evaluation zwischen zwei Fällen unterschieden. Zum einen ist es möglich, dass die Sensorfrequenz niedriger als die Frequenz der Zyklen ist und zum anderen ist es möglich, dass die Frequenz des Sensors höher als die der Zyklen ist. Als Messpunkt wurde hierbei die Zeit gewählt, die ein neues Element warten muss, bis es das Kontextmodell bekommt und aktualisieren kann. Diese ergibt sich jeweils pro Aktualisierungs-Operator von dem Abschicken eines Requests bis zur Antwort durch den Broker anhand des Kontextmodells.

Die Tests fanden auf einem Intel Core 2 Duo mit 2 CPUs bei 2.20 GHz und 4 GB Arbeitsspeicher statt. Als Betriebssystem wurde Windows 7 mit 64 Bit verwendet.

6.1 Niedrigere Sensorfrequenz

Wie zuvor beschrieben beträgt die Verarbeitungszeit von Zyklus A $500\ \mu s$ und von Zyklus B $1000\ \mu s$. Dementsprechend wurde für die Sensoren eine kleinere Frequenz genommen, sodass die Sensoren jede $2000\ \mu s$ ein neues Element erzeugen. Die Ergebnisse des Experiments zeigt Abbildung 6. In der Abbildung ist zu sehen, dass die durchschnittliche

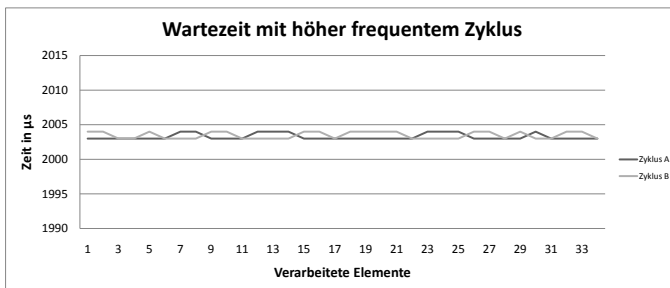


Abbildung 6: Wartezeit bei einem niedriger frequenten Sensor

Wartezeit eines Elements bei etwas über $2000\ \mu s$ liegt. Die Verarbeitungsgeschwindigkeit wird demnach von dem Sensor, also der kleineren Frequenz, vorgegeben. Da die Zyklen mit einer größeren Frequenz arbeiten, ist die Aktualisierung bereits beendet bevor ein neues Element das System erreicht. Der Broker muss bei einem Request von Zyklus A erst warten, bis ein Request von Zyklus B eingegangen ist. Dies ist nötig, damit der Broker weiß, dass nicht eventuell der Request von Zyklus B vor dem von Zyklus A abgearbeitet werden müsste. Entsprechend richtet sich die Wartezeit eines Elements nach der höchsten

Frequenz der Sensoren. Da hier die Frequenz der Sensoren konstant ist, ist auch die Wartezeit aller Elemente konstant. Diese schwankt lediglich ein wenig auf Grund des Scheduling und der momentanen Auslastung des Gesamtsystems. Um hierbei nicht von der Frequenz einer Datenquelle abhängig zu sein, kann ein Heartbeat-Mechanismus in Form von Punctuations [TMSF03] verwendet werden. Punctuations sind in der Regel einfache Zeitstempel, die den zeitlichen Fortschritt in einem Datenstrom angeben. Ein Operator kann dann davon ausgehen, dass nach einer Punctuation nur noch Elemente folgen, die einen größeren Zeitstempel als den Zeitstempel der letzten Punctuation haben. Da dies der Semantik des Minimum-Zeitstempels entspricht, wie er in Abschnitt 5.2 verwendet wird, ist es mit einer Punctuation möglich, den Minimum-Zeitstempel bereits vor einem neuen Messwert zu berechnen und somit die Wartezeit für einen anderen wartenden Messwert zu verkürzen.

6.2 Höhere Sensorfrequenz

Im Gegensatz zum vorherigen Test, wurde des Weiteren der Fall betrachtet, in dem die Sensorfrequenz höher ist. Indem die Sensoren in einem Abstand von $100\mu s$ neue Elemente erzeugen, wurde eine größere Frequenz als die der Zyklen gewählt. Abbildung 7 zeigt die Messergebnisse. Hierbei ist zu erkennen, dass die Wartezeit einzelner Elemente

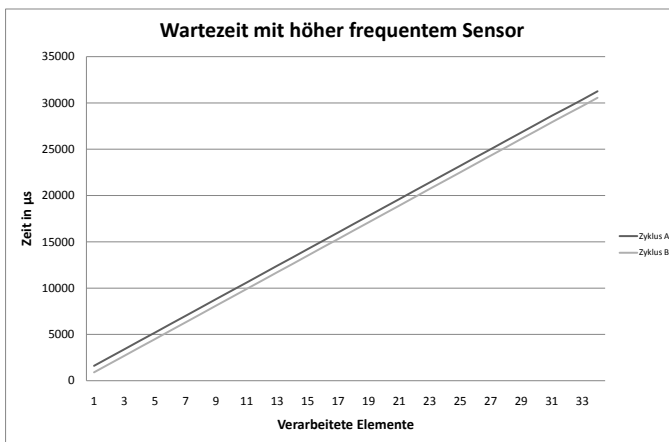


Abbildung 7: Wartezeit bei einem höher frequenten Sensor

bei jedem weiteren Element zunimmt. Dies begründet sich dadurch, dass neue Elemente wesentlich schneller beim System ankommen, als sie vom System verarbeitet werden können. Da die Frequenz der Sensoren konstant ist, nimmt auch die Wartezeit je Element mit einem konstanten Faktor zu, sodass die entsprechend konstante Steigung entsteht. Die ansteigende Wartezeit entspricht ebenfalls der Aktualität des Kontextmodells. Je länger ein Element auf die Aktualisierung warten muss, desto mehr weicht auch das aktuell gültige Kontextmodell von der tatsächlichen Welt ab. Um hierbei ein möglichst aktuelles Kon-

textmodell zu gewährleisten, ist es sinnvoll, dass für die Sensoren im Vergleich zu der Verarbeitungsfrequenz der Zyklen eine kleinere Frequenz gewählt wird.

Die Experimente zeigten demnach die erwarteten Ergebnisse, indem bei niedrigerer Frequenz eine bestimmte Zeit gewartet werden muss. Hierbei kann man u.a. beobachten, dass ein Messwert aus Sensor A erst bedient werden kann, wenn ein weiteres Element aus Sensor B vorhanden ist. Denn erst zu diesem Zeitpunkt kann der Broker sicher gehen, dass aus Sensor B nicht eventuell noch Messwerte folgen, die zuerst ausgeführt werden müssten. Demnach entstehen zusätzliche Latenzen, die auf Grund der Transaktionskontrolle des Brokers entstehen und bei einer nicht-transaktionssicheren Implementierung mit einem (Hauptspeicher-)DBMS entsprechend nicht vorhanden wären. Des Weiteren entsprechen die Ergebnisse bei einer höheren Sensorfrequenz ebenso den Erwartungen, indem es einen Systemüberlauf gibt, da wie beschrieben zusätzliche Latenzen entstehen.

7 Fazit

Obwohl sich die Verwendung eines DSMS im Bereich kontextsensitiver Anwendungen anbietet, werden für die Verarbeitung von Sensordaten in der Regel feste Programme eingesetzt. Möchte man jedoch solche Konzepte in einem DSMS umsetzen, muss man auch die aktuelle Umgebung in einem Kontextmodell abbilden. Hierbei muss unter anderem eine möglichst effiziente Speicherung berücksichtigt werden. Bei den Aktualisierungen des Kontextmodells, die jeweils periodisch durch ein neu ankommendes Element angestoßen werden, müssen die verschiedenen lesenden und schreibenden Zugriffe auf das Kontextmodell berücksichtigt werden, damit Aktualität und Konsistenz des Kontextmodells sichergestellt werden. Wir haben dazu den Broker-Operator auf Grundlage einer festen Semantik eingeführt und eine abstrakte Implementierung gezeigt. Dieser Operator erlaubt es, das Kontextmodell transaktionssicher im Hauptspeicher zu verwalten, ohne dass dabei die temporale Ordnung der Daten vernachlässigt wird. Die Umsetzung des Brokers erfolgte in dem komponentenbasierten Datenstrom Management Framework Odysseus. Diese Umsetzung wurde ebenso für Experimente genutzt. Hierbei wurde das Verhalten bei verschiedenen Frequenzen von Datenquelle und Zyklen beobachtet. Bei einem langsameren Sensor erfolgen Aktualisierung innerhalb einer konstanten Zeit, die auf Grund der Transaktionskontrolle nicht nur vom eigenen Sensor sondern auch von allen Sensoren beeinflusst wird. Werden die Daten jedoch schneller erzeugt, als sie im Zyklus verarbeitet werden können, so entsteht ein Datenstau, wodurch Aktualität des Kontextmodells und die tatsächliche Umgebung mit der Zeit immer mehr auseinander laufen.

Zukünftig wird der Broker-Operator in einem DSMS-basierten Framework für FAS eingesetzt. Dabei werden neben der zeitlichen Koordination insbesondere auch anwendungsspezifische Löschrategien für das Kontextmodell entwickelt, um ungültige Elemente aus dem Kontextmodell zu entfernen. Außerdem werden Anfragepläne mit zwei oder mehr Broker-Operatoren entwickelt, um neben dem eigentlichen Kontextmodell auch temporäre Kontextmodelle zuzulassen. Dies ist insbesondere deswegen wichtig, weil auf Grund möglicher Fehlmessungen nicht jedes Objekt direkt in das eigentliche Kontextmodell aufgenommen werden soll, sondern erst nachdem es mehrmalig erkannt worden ist.

Literatur

- [AAB⁺05] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong Hwang, Wolfgang Lindner, Anurag Maskey, Alexander Rasin, Esther Ryzkina, Nesime Tatbul, Ying Xing und Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, 2005.
- [ABB⁺03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein und Jennifer Widom. STREAM: the stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, Seiten 665–665, San Diego, California, 2003. ACM.
- [ACc⁺03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul und Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.
- [BGJ⁺09] A. Bolles, M. Grawunder, J. Jacobi, D. Nicklas und H.-J. Appelrath. Odysseus: Ein Framework für maßgeschneiderte Datenstrommanagementsysteme. In *39. GI Jahrestagung, Workshop: Verwaltung, Analyse und Bereitstellung kontextbasierter Informationen*, 2009.
- [CCMP06] C Cappiello, M Comuzzi, E Mussi und B Pernici. Context Management for Adaptive Information Systems. *Electronic Notes in Theoretical Computer Science*, 146(1):69–84, 2006.
- [CEB⁺09] Nazario Cipriani, Mike Eissele, Andreas Brodt, Matthias Grossmann und Bernhard Mitschang. NexusDS: a flexible and extensible middleware for distributed stream processing. In *IDEAS '09: Proceedings of the 2009 International Database Engineering & Applications Symposium*, Seiten 152–161, New York, NY, USA, 2009. ACM.
- [CGM09] Badrish Chandramouli, Jonathan Goldstein und David Maier. On-the-fly Progress Detection in Iterative Stream Queries. *Proceedings oth the VLDB Endowment 2009*, 2(1):241–252, 2009.
- [DS00] Guozhu Dong und Jianwen Su. Incremental maintenance of recursive views using relational calculus/SQL. *SIGMOD Rec.*, 29(1):44–51, 2000.
- [DSTW02] J.P. Dittrich, Bernhard Seeger, D.S. Taylor und Peter Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proceedings of the 28th international conference on Very Large Data Bases*, Seite 310. VLDB Endowment, 2002.
- [EN09] Ramez A. Elmasri und Shamkant B. Navathe. *Grundlagen von Datenbanksystemen*. Pearson Studium, 3. aktualisierte auflage. bachelorausgabe.. Auflage, 2009.
- [GADI08] D. Gyllstrom, J. Agrawal, Y. Diao und N. Immerman. On supporting kleene closure over event streams. In *IEEE 24th International Conference on Data Engineering, 2008. ICDE 2008*, Seiten 1391–1393, 2008.
- [GAE06] Thanaa M. Ghanem, Walid G. Aref und Ahmed K. Elmagarmid. Exploiting predicate-window semantics over data streams. *SIGMOD Rec.*, 35(1):3 – 8, 2006.

- [GAW⁺08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu und Myungcheol Doo. SPADE: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Seiten 1123–1134, Vancouver, Canada, 2008. ACM.
- [GBz06] Lukasz Golab, Kumar Bijay und M. Özsu. On Concurrency Control in Sliding Window Queries over Data Streams. In Yannis Ioannidis, Marc Scholl, Joachim Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust und Christian Boehm, Hrsg., *Advances in Database Technology - EDBT 2006*, Jgg. 3896 of *Lecture Notes in Computer Science*, Seiten 608–626. Springer Berlin / Heidelberg, 2006.
- [HJ04] X. Huang und C.S. Jensen. Towards a streams-based framework for defining location-based queries. In *Proc. STDBM*, Seiten 78–85. Citeseer, 2004.
- [Krä07] Jürgen Krämer. *Continuous Queries over Data Streams - Semantics and Implementation*. Dissertation, Philipps-Universität Marburg, 2007.
- [KS05] Jürgen Krämer und Bernhard Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *Proceedings of the 11th International Conference on Management of Data (COMAD)*, Seiten 70–82, 2005.
- [MA08] M.F. Mokbel und W.G. Aref. SOLE: scalable on-line execution of continuous queries on spatio-temporal data streams. *The VLDB JournalThe International Journal on Very Large Data Bases*, 17(5):995, 2008.
- [PS04] K. Patroumpas und T. Sellis. Managing trajectories of moving objects as data streams. In *Proceedings of the STDBM*, Jgg. 4. Citeseer, 2004.
- [Rea10] Realtime Monitoring GmbH. RTM Analyzer, 2010.
- [RHC⁺02] Manuel Roman, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell und Klara Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.
- [See04] Bernhard Seeger. Datenströme. *Datenbank-Spektrum*, 4(9):30–33, 2004.
- [TMSF03] P.A. Tucker, D. Maier, T. Sheard und L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15:555–568, 2003.
- [WRML08] Mingzhu Wei, Elke Rundensteiner, Murali Mani und Ming Li. Processing recursive XQuery over XML streams: The Raindrop approach. *Data Knowl. Eng.*, 65(2):243–265, 2008.

Tracking Hot-k Items over Web 2.0 Streams[◇]

Parisa Haghani * Sebastian Michel [‡] Karl Aberer *

* Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
parisa.haghani@epfl.ch, karl.aberer@epfl.ch

[‡] Universität des Saarlandes, Saarbrücken, Germany
smichel@mmci.uni-saarland.de

Abstract: The rise of the Web 2.0 has made content publishing easier than ever. Yesterday's passive consumers are now active users who generate and contribute new data to the web at an immense rate. We consider evaluating *data driven* aggregation queries which arise in Web 2.0 applications. In this context, each user action is interpreted as an event in a corresponding stream e.g., a particular weblog feed, or a photo stream. The presented approach continuously tracks the most popular tags attached to the incoming items and based on this, constructs a dynamic top-*k* query. By continuous evaluation of this query on the incoming stream, we are able to retrieve the currently *hottest* items. To limit the query processing cost, we propose to pre-aggregate index lists for parts of the query which are later on used to construct the full query result. As it is prohibitively expensive to materialize lists for all possible combinations, we select those tag sets that are most beneficial for the expected performance gain, based on predictions leveraging traditional FM sketches. To demonstrate the suitability of our approach, we perform a performance evaluation using a real-world dataset obtained from a weblog crawl.

1 Introduction

The world has turned into one large-scale interconnected information system with millions of users. End users, with the advent of Web 2.0, are now content generators who actively contribute to the Web. User generated data is usually in form of semi-structured text like personal blog entries with categorization ¹ or images and videos annotated with tags [Fli, You]. Each user action, for example uploading a picture, tagging a video or commenting on a blog, could be interpreted as an event in a corresponding stream. Data stream processing has gained a lot of attention in the recent years (see [BBD⁺02, Mut05] for surveys), since many of today's applications are best captured in this model. Data items in different formats stream in to a processing unit where each item has the chance of being

[◇] This work is partially supported by NCCR-MICS (grant number 5005-67322), the FP7 EU Project OKKAM (contract no. ICT-215032), and the German Research Foundation (DFG) Cluster of Excellence "Multi-modal Computing and Interaction" (MMCI).

¹ <http://google.blogspot.com/>, <http://www.weblogs.com/>, <http://www.blogger.com/>

seen once before being archived for later uses. While this model has been successfully applied in scenarios such as sensor networks, traffic monitoring and financial data feeds, Web 2.0 generated data has less frequently been treated as streams. Most data mining approaches on this ever growing source of data run their analysis algorithms in an offline fashion [KNRT05, Kle02, HJSS06], hence disregarding the live nature of the web.

Given the immense volume of data being published on the web and the desire of consuming newly published data, there is an increasing need for processing this information in real time in efficient ways. All this gives rise to considering this data in a streaming model.

As an *example* of temporal streams of information in a Web 2.0 application consider published content in form of news articles or posts on personal weblogs (blogs). Explicit temporal annotations (i.e. *written at*, *uploaded at*) of the content of weblogs or news portals makes them natural items of a temporal stream. Mechanisms such as RSS and atom are used to notify users of newly published data on their favored weblogs or news portals. The items in a blog feed stream are generated at distributed sources depending on the subscriptions which are made by the user. The large body of information retrieval techniques can be used in order to extract categories or topics from the published text [APL98, ACD⁺98].

We consider online processing of aggregation queries over streaming data where each data item carries a particular set of tags with it. We aim at monitoring the hottest items at each time by defining a top- k query of currently popular tags. Hot items are subsequently defined as those items which have high score with regard to the defined query.

1.1 Problem Statement and Contribution

We consider a stream \mathcal{S} of tagged items where each item has the following format:

$$d = \langle itemId, time, \mathcal{T}_d \rangle$$

$itemId$ is a unique identifier specifying the object this item is describing, i.e. URL of an image or post, and $time$ represents the time when d was produced. Let $\mathcal{T} = \{t_1, \dots, t_n\}$ be the global set of tags which are used to annotate items. $\mathcal{T}_d \subset \mathcal{T}$ is the set of tags with which d is annotated. The number of tags an item carries is usually very small (e.g., around 5) compared to standard document retrieval where a text document contains lots of terms. For each tag we assume a given score $score(d, t)$ that reflects the relatedness of the item to the tag.

We further assume *in-order* streams; items arrive in the same order that they are generated. In most streaming scenarios, as well as ours, recent items are of more interest than old ones. This is captured by the sliding window model. A sliding window (W) is assumed over the stream and items are considered *valid* while they belong to this window. Sliding windows can be either count or time based, i.e., bounding the number of items either by count or focusing only on those that occurred in a particular time interval.

At each point in time we can compute statistics over the tags used in items currently in the sliding window W or compute aggregation queries over these items. This view forms the basis of our approach, which builds on statistics on tag usages to determine a set of

popular tags. This tag set is then interpreted as a continuous and dynamic keyword query which is executed against the sliding window as time evolves. We call this query dynamic as it is re-build with evolving time due to changes in tag popularities.

Definition 1 Hot Tags and Hot Items: *At each timestamp τ , the set of hot tags (H_τ) consists of the c tags with the highest popularity.*

The set of hot tags defines the query we use to rank the valid items, i.e., the query is data-driven and changes with time as the popularity of tags changes. For a valid item, we define its current score as the sum of scores of the hot tags it carries. More formally,

$$s(d, H_\tau) := \sum_{t \in H_\tau \cap T_d} \text{score}(d, t)$$

The task is to continuously compute the top- k items as the query changes. In contrast to standard top- k query processing over text (or XML) documents, here, the query is supposed to be rather big to capture not only a few but many hot topics for diversity reasons. In summary, the considered tags (features, in standard IR terminology) is small whereas the query is long, which is in clear contrast to traditional query processing techniques.

In this work we focus on efficiency aspects and the potential of pre-aggregations and how to decide which subqueries to pre-compute. For the actual decision which tags should be considered in as query terms, one can think of other measures than the pure popularity count based methods we use in our work, e.g., methods that aim at identifying trending (hot) topics.

In this paper we make the following contributions. We show how to continuously compute the set of hot items over social (Web 2.0) data streams by defining a dynamic top- k aggregation query and show how pre-aggregations of popular sub queries can be used to efficiently process the query. We evaluate our proposed methods on a real-world dataset of blog posts showing the suitability of our approach.

This paper is organized as follows. Section 2 presents the related work. Section 3 briefly describes the general structure that we consider in this paper together with a baseline algorithm. Section 4 describes the problem of pre-aggregating groups of index lists for efficient query processing and presents next to an offline problem definition an efficient and effective approximation for online processing. Section 5 presents the experimental evaluation. Section 6 concludes the paper.

2 Related Work

Data stream processing has been a hot topic in the past years as many of today's applications require real-time processing of dynamic data. For comprehensible surveys of this topic in general see [BBD⁺02, Mut05]). Early works mostly consider one-pass algorithms in limited space over the whole stream where all tuples are considered valid at all times. A related problem to ours is reporting on *quantiles* or *heavy hitters* in streams. The goal

is to report on most repeated items in the stream, when the number of items is so high that keeping statistics for each is not possible. Approximate solutions to this problem exist which make use of techniques such as the famous AMS sketches [AGMS02], or more recently group testing, see for example [CCFC04, CM03] and the references within. In our work, the number of tags we consider and desire to know the hottest amongst is small enough such that exact statistics could be kept for each.

Another line of research in stream processing is dedicated to top- k query answering in data streams. Mouratidis et al. [MBP06] maintain a skyline [BKS01] which represents the possible top- k candidates. Their solution is optimized for *fixed* queries and they focus on changes introduced by items timing out or new items arriving in. In a more general setting, [DGKS07] proposes indexing methods for answering *ad hoc* top- k queries based on arrangements. While our queries can not be considered as *fixed* (as the set of hot tags changes over time with new items arriving) they are not completely *ad hoc* either. We exploit this fact to pre-aggregate parts of the query which can be used several times in future queries. Jin et al. [JY⁺08] consider top- k queries on uncertain streams where the data items are associated with existential probabilities. In our envisioned applications all items are certain.

Mainly motivated by the wealth of news feeds and other online information streams, another related problem is *Topic Detection and Tracking* (TDT) which has been extensively studied in the past few years [APL98, ACD⁺98, HCL07]. The goal here is to detect new events appearing in the data stream and tracking those events in order to later identify data which further discuss the same event. Another related topic is mining frequent itemsets in a data stream. In a recent work Calders et al [CDG07] define a new measure as the frequency of an itemset and propose an incremental algorithm that allows for reporting the exact frequencies of frequent itemsets. The problem of itemset mining is orthogonal to our problem and can be used to improve the quality of our choice of pre-aggregation queries. In another line of research related to Web 2.0 applications with temporal considerations, Hotho et al. [HJSS06] consider discovering topic-specific trends in folksonomies which are collections of resources tagged by users (such as Flickr or del.icio.us²). Their analysis is based on the famous PageRank algorithm. They perform the algorithm in an offline manner and assume the whole corpus of data to be available. Weblog evolution is considered in [KNRT05], where *time graphs* are introduced and used for community tracking again in an offline mode. In [MK09], the goal is to identify weblogs defined as *starters* and *followers* specified by certain linking relations in an efficient way. In contrast to the above, we continuously evaluated the data as it arrives in an online manner. For a survey of temporal data analysis methods see [Kle06] and the references within.

Keeping the query results updated as data streams in with high rates requires high performance evaluation of top- k queries. One way to improve the performance of expensive queries is to maintain their results as materialized views. In order to avoid reprocessing a top- k query in face of updates in the database, such as insertions or deletions, authors in [YYY⁺03] suggest maintaining a top- k' view, where $k' > k$ and show how to choose k' dynamically to adapt to the system workload. In [HKP01] authors investigate answering a top- k query based on the materialized results of another top- k query where the preference

²<http://del.icio.us>

function is a linear combination of all attributes of tuples. It is shown how to decide given a preference function and its top- l results if the top-1 result of another preference function can be found in these materialized l tuples. In [DGKT06], the TA algorithm is adapted to the case where a set of views, not necessarily the single inverted lists, are available. The views are visited in a lock-step manner and in each iteration the maximum score of unseen tuples are calculated by a linear programming optimization, given the preference functions of each of the views. Given a set of views, the best subset for answering a query is chosen based on a process simulating the TA utilizing the data distributions in each view. In the same line, [KPSV09], investigate top- k query processing when intersection of single inverted lists are also available. A combinatorial solution is proposed to solve the specific linear program appearing when the set of lists consist of only single or intersection of two single lists. A very interesting result of the paper is that in order to guarantee instance optimality all available lists should be investigated. In a streaming scenario however, maintaining the intersection of all pairs of single lists is not possible due to memory constraints. In this work we propose to maintain the intersection of several lists instead of just pairs of them and we chose the intersections based on the benefits they potentially have for future data-driven top- k queries.

3 System Model and Structure

In this section we briefly describe the general structure that we consider. As mentioned in Section 1.1 we consider one data stream as the input to our system where the items in this stream contain a list of tags and they are considered valid while belonging to a sliding window.

We assume all valid items are sorted in a first-in-first-out list. This provides an efficient mechanism for evicting expired items. Newly arriving items in the stream are placed at the head of this list and old items are dropped from the tail. In addition to the time sorted list, we maintain a hash index on the valid items that point to the set of their tags. Furthermore, for each tag, we keep a sorted list of items that have been annotated with this tag. Let l_i represent the list maintained for tag t_i . l_i is sorted based on t_i 's score for each item in descending order. When an item expires, it is also removed from the sorted list it belongs to. Considering newly arriving items is easily achievable as it causes only insertions to a few lists plus one insertion to the hash index and the time sorted list, as described above. Note that as opposed to standard top- k processing where each document has potentially very many features (terms), here, the average number of tags per item is rather small. As a result updating the structures with new arrivals does not incur high cost.

For basic query execution we employ the threshold algorithm (TA) [Fag02], which works as follows. It reads in parallel from the index lists, which are sorted by score in descending order. For each item observed it looks up its score in all other lists it has not been observed so far, which is done in our case with one lookup to the hash map as described in the previous paragraph. The aggregated scores of the items at the current sequential access scan depth define the stopping condition. The computation can be stopped if there are at least k items with a score better than the aggregated score at the sequential scan lines. We

employ the TA algorithm over the single term index lists as our baseline algorithm.

The top- k query needs to be re-evaluated in two cases: first, when an item which was part of the top- k results expires. The second case happens when the set of top tags changes and causes a change in the query aggregation function. In order to avoid re-computations from scratch when a hot item expires, a k -skyband over the score-time space can be kept [MBP06]. The k -skyband of a query contains only those items which have a chance of becoming a top- k result during their life time. When an item which was part of the top- k results expires, it is enough to evaluate the query on the k -skyband, instead of the entire valid items, to fill in the top- k results. This dramatically decreases the cost of re-evaluations, however, it is only useful when the query remains unchanged. For the rest of this paper we do not consider possible optimizations when the top- k query is not changing, as this is a well addressed problem [MBP06, DGKS07], rather, we will focus on solutions for the changing query issue. In the next section we describe our approach for pre-aggregating stable parts of the top- k query in order to decrease the cost of evaluations when the query changes.

4 Grouping for Pre-Aggregation

Observing the changes in the top- k query itself, which is considered to be quite large (~ 100 tags), shows that although the query itself changes more or less every time, there is a fraction of tags that remain as part of the query for a long duration of time. These consists of those tags which are popular most of the time and represent current long-lived events. Observing stable sub-queries, motivates us to maintain pre-aggregations for those sub-queries which can later be used to evaluate the complete query more efficiently.

In this section we propose to group lists corresponding to “stable” tags together to reuse their aggregated results. More precisely, we pre-aggregate certain lists and try to assemble at query time the final top- k result given the pre-aggregated values.

4.1 Optimal Solution

To better understand the complexity of the problem, in this section we formulate an offline algorithm. The offline algorithm assumes a finite stream and complete knowledge over incoming data. Therefore the set of different top- k queries for a given time period is known to the offline algorithm.

Given a set of queries $\mathcal{Q} = \{Q_1, Q_2, \dots, Q_n\}$, the goal is to find an optimal set of subsets of tags \mathcal{S} that can answer all queries in \mathcal{Q} efficiently, re-using pre-aggregations in \mathcal{S} . Each member of \mathcal{S} is a subset of tags and if its cardinality is larger than one, represents a pre-aggregation of the lists maintained for the tags it contains. For example $\mathcal{S} \ni S_i = \{t_j, t_k\}$ means we are maintaining a sorted list for $t_j \vee t_k$. Let L_i represent the list corresponding to S_i . Items in L_i are sorted based on their score with regard to S_i : $s(d, S_i) := \sum_{t \in S_i \cap T_d} \text{score}(d, t)$.

In case of ties, more recent items are preferred. L_i is created utilizing the simple lists we maintain for tags which are members of S_i . Assuming equal length l for all simple lists, the cost of aggregating k such lists is $k * l$.

Now assume a query Q_y . Recall that each query is specified by a set of tags. We say a subset $S'_y \subset \mathcal{S}$ *exactly covers* Q_y if members of S'_y are pairwise disjoint and $\bigcup_{S_i \in S'_y} S_i = Q_y$. If a subset exactly covers a query, a standard TA algorithm can use it to evaluate that query. The effectiveness of a list L_i depends on the co-occurrences of tags in S_i in the stream of items. We assume the percentage of items likely to be read before TA can stop is known for a list L_i and we denote it by c_i . Note that c_i depends on the query and other available lists, but for simplicity we consider it as an independent fixed value. The cost of evaluating a query Q_y using S'_y can be estimated by: $\sum_{S_i \in S'_y} c_i$.

Let \mathcal{P} be the powerset of $\bigcup Q_i$. Given the above cost functions, we can formulate our goal as an optimization problem which aims at minimizing the following cost function with regard to the boolean variables x_{ij} :

$$\sum_{S_i \in \mathcal{P}} y_i * |S_i| * l + \sum_{Q_j \in \mathcal{Q}} x_{ij} * c_i$$

and the following constrains:

$$\begin{cases} y_i = \bigvee_j x_{ij} & (C1) \\ \forall Q_j \forall t \in Q_j \sum_{i: t \in S_i} x_{ij} = 1 & (C2) \end{cases}$$

$x_{ij} = 1$ shows that S_i is used in evaluating Q_j . $y_i = 1$ if S_i is used in evaluating at least one query. The first constraint (C1) assures this. The first summation in the cost function accounts for the pre-aggregation expenses while the second part shows the evaluation cost. The second constraint (C2) ensures that the set of S_i 's used for evaluating each query exactly cover that query.

The above optimization problem is not a standard linear programming problem, as the variables y_i depend on x_{ij} 's. However, even if we ignore the first part of the cost function (the query evaluation cost), we face a 0-1 linear programming problem which is known to be NP-hard (cf., e.g., [MS08]).

4.2 Efficient Grouping

Given the complexity of the problem described above and the fact that the set of future top- k queries is actually not known in advance, we address the problem with an approximate approach.

Clearly it is beneficial to pre-aggregate sets of tags which frequently appear in the future top- k queries: Aggregating the corresponding lists of a set of tags pays off only when the resultant list can be used enough number of times in future queries. For each observed

tag we maintain the number of times it has appeared in the set of hot tags and predict its probability of being part of the aggregation query based on this past information.

Assume the number of single tags with probability of appearing in future queries larger than a specific threshold is r . These tags have to be grouped together to form a pre-aggregated list. However, grouping all of them together may not be beneficial, as to be able to use such a pre-aggregation *all* involved tags should be part of the query. The probability that a pre-aggregation of m single lists is usable in future queries, decreases with increasing m : if p is the probability of the most frequent tag, and assuming tags appear independent of one another, p^m is an upper bound of the probability that this aggregation list is usable. We should therefore, pre-aggregate *subsets* of the r candidate tags.

Grouping those tags which co-occur together in the streaming items is highly beneficial for the overall performance as they have higher chances of appearing *together* in future queries. Given the data-driven nature of the query, the query evaluation using the TA algorithm can be done more efficiently due to the already pre-aggregated partial queries. A pre-aggregation of tags which do not co-occur together and aggregating them creates a list of size of sum of the sizes of single lists with non-aggregated scores. On the other hand, aggregating single lists which have high correlation, i.e., their corresponding tags occur together, results in a list with more score variations (in case of ties in the original list) and higher scores, which is more effective in decreasing the threshold value maintained by the TA algorithm and causing it to stop reading more entries.

As a measure of tag co-occurrence we calculate the resemblance value for two index lists, which is defined as the fraction of the size of their intersection over the size of their union. Based on the given intuitions above, in the next section we describe our proposed algorithm for selecting tag sets to be materialized.

4.3 Tag Set Generation

To actually compute the tag sets to be materialized our algorithm considers all tags that frequently occurring in the queries with a probability above a parameter α . Since the cardinality of the set of tags is not large, we can maintain exact statistics for the number of occurrences of each tag in a query. We normalize the number of occurrences and use it as the probability of tag's occurrence in future queries.

Based on this group of tags we generate the tag sets of interest in the following way, illustrated in Figure 1:

1. each tag is considered to be a node of a graph
2. for each pair of tags the resemblance is calculated
3. each pair of tags with resemblance $\geq \rho$ is treated as an edge in a graph
4. the connected components of the graph are sets of tags to be materialized

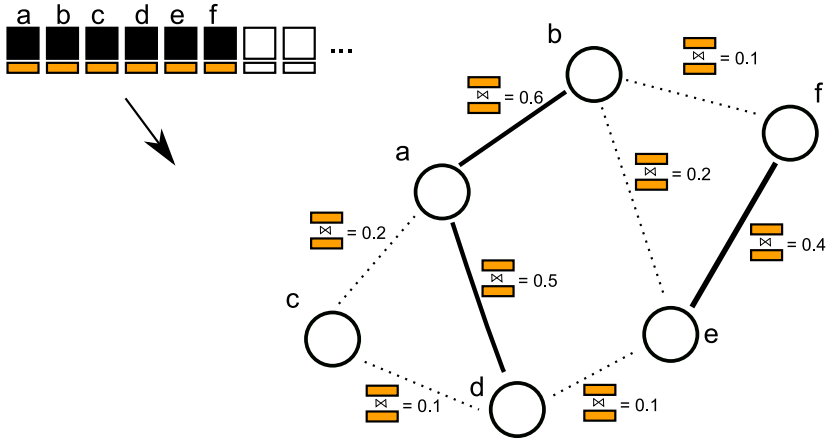


Figure 1: Illustration (showing a subset of all cases to be considered) of the process of determining tag sets of interest for pre-aggregation for $\rho = 0.3$. Given the top re-occurring tags as the nodes in a graph, we connect those nodes whose index lists have a resemblance of at least 0.3. All resulting connected components are then selected and the corresponding index lists pre-aggregated.

This technique favors those frequently reoccurring parts of the query that also frequently appear *together* in the data stream.

4.4 FM Sketches for Resemblance Calculation

As the computation of the exact resemblance is extremely expensive we employ a sketching technique that can efficiently estimate the resemblance value independent of the size of the involved index lists. In addition, as even exact resemblance numbers cannot guarantee the optimal pre-aggregation, the effect of slightly inaccurate resemblance numbers are negligible.

We make use of the well known Flajolet-Martin sketches (FM sketches) [FM85], which are compact and precise estimators of the cardinality of a multi-set. Given two sets S_1 and S_2 and their corresponding synopses in form of FM sketches, one can determine the size of the intersection by combining the sketches in an extremely efficient bit-wise fashion. More precisely, one obtains actually the size of the union given the bit-wise OR operation of the bit-sets of the two sketches. Then, the size of the intersection is given by the inclusion-exclusion principle ($|S_1 \cup S_2| = |S_1| + |S_2| - |S_1 \cap S_2|$), hence we can estimate the resemblance value.

As we keep index lists for the tags we observe, there is only the small overhead of maintaining a sketch for each of these lists. When enumerating the candidate tag sets we estimate their suitability to the query processing based solely on the sketches. There is no need to compute the aggregation and assess its size as the size is directly given by the

sketch combinations, which is very efficient.

Due to the inherently approximate nature of the sketches, the resemblance values are not exact, which leads to decisions of which tag sets to materialize that varies from the algorithm employing the true resemblance numbers.

5 Experiments

We have implemented our algorithm in Java 1.6 and executed on a Windows 2003 server with a quad core 2.33 GHz Intel Xeon CPU, 16GB RAM, and a 800GB RAID-5 disk.

We have obtained the ICWSM 2009 Spinn3r Blog Dataset³. It consists of 44 million blog posts between the time period of August 1st and October 1st, 2008. Each blog entry (post) consists of plain text, a timestamp, a set of tags, and other meta information such as the blog's homepage URL etc. The data is formatted in XML and is further arranged into tiers approximating to some degree search engine ranking. We have parsed the blog posts for the highest tier levels resulting in 11,395,571 (timeStamp, postId, tags)-entries, with 2,444,780 distinct postIds, hence, an average of ~ 2.2 tags per blog entry. For the score of a document w.r.t. a particular tag we simply consider score 1 if the tag is attached to the document, 0 otherwise. While in principle a measure like *tag frequency* would be more suitable, the way the dataset is generated limits us to the boolean values.

Algorithms

We consider the performance of three algorithms in this experimental evaluation. All are based on the TA algorithm [Fag02]. The difference stems from the index lists they can involve in the query processing. More precisely, we run the following algorithms:

- **plain:** This is the plain algorithm involving only accesses to single-tag index lists.
- **comb:** This algorithm uses pre-aggregation of tag sets that are supposed to help the query execution. The set of tags to be pre-aggregated are chosen using the algorithm described in Section 4.3. True resemblance values are calculated by merging lists and measuring the resultant size.
- **combsketch:** This algorithm also uses pre-aggregation tag sets as described in Section 4.3. However, the resemblance values are estimated using sketches as described in Section 4.4.

Note that the comb algorithm is in fact impractical, as it incurs huge costs just for measuring the resemblance values. However we ignore this cost and use this algorithm to show the best achievable performance using our proposed set aggregation method.

³<http://www.icwsml.org/2009/data/>

Measures of Interest

We will report on several measures as part of our performance study. Note that we do not report on accuracy measure as *all algorithms report the exact top-k results* to the query described above. We consider the number of entry accesses as the main cost to assess the suitability of the methods under comparison. We split this measure up in several ingredients to better understand the strong and weak points of the approaches. In particular for the algorithms that use pre-aggregation, the cost for materializing lists for sets of tags does not occur in each query processing step. We measure:

- **eval_cost:** This measure reports on the average number of entry accesses the threshold algorithm makes to calculate the results.
- **pre-aggregation_cost:** With this measure we provide an insight on how costly the pre-aggregation operation is, that means, how many entries on average need to be accessed when materializing the index lists for sets of tags, determined by the selection algorithm. The plain algorithm does not incur any pre-aggregation cost.
- **total_cost:** In addition to the measures described above we also report on the total cost which consists of the total (non-averaged) cost for all query evaluations plus the overall cost for doing the pre-aggregation. We ignore the cost for calculating the resemblance values.

Results

We run the mentioned three algorithms for different parameter settings averaging over 45 query evaluations for each setting. The query evaluation is fired at every 500 items. The tag set generation algorithm (described in Section 4.3) is run periodically at every 20 evaluations. Unless otherwise stated we use a time-based sliding window of size $W = 10,000,000$ milliseconds. The default number of desired top- k items denoted by $kdocs$ is 100. The number of tags used in defining the query is denoted by $ctags$ and its default value is set to 75.

We first observe the effects that parameters α and ρ have on the costs incurred by our proposed algorithms. Figures 2,3, and 4 shows the different cost values while varying the parameter α and fixing all other parameters. As explained in Section 4.3, α denotes the threshold for considering a tag for the subsequent tag set generations. Figure 2 presents the evaluation cost with changing α . Small α values causes the algorithm to consider tags which actually do not occur later in the query. These tags may have high enough resemblance with other tags to be part of a connected component. The tag set corresponding to such a component is however, useless, since it contains a tag which does not actually appear in the query. As a result, both comb and combsketch have total costs close to plain with small α values. On the contrary, for large enough values of α a large fraction of materialized lists are actually reusable, therefore the evaluation cost of comb and combsketch is much smaller than plain. For too high values of α , less than necessary number of

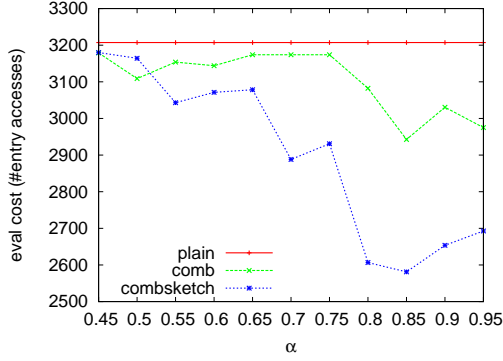


Figure 2: Eval_cost values when varying the α parameter. $W=10,000,000ms$, $kdocs=100$, $ctags=75$, $\rho = 0.6$

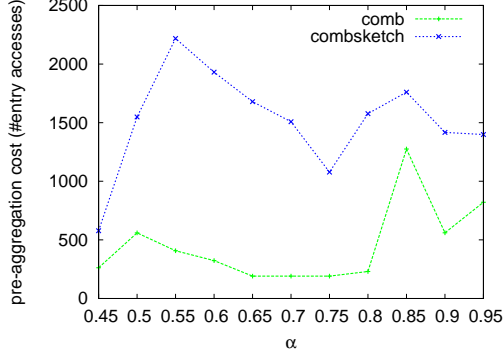


Figure 3: Pre-aggregation_cost values when varying the α parameter. $W=10,000,000ms$, $kdocs=100$, $ctags=75$, $\rho = 0.6$

tags are actually considered, lowering the total benefits of them in evaluating the queries. The pre-aggregation_cost is shown in Figure 3. We see that for $\alpha = 0.85$ both comb and combsketch have high pre-aggregation_cost which actually pays off very well, as the total cost at this value has a minimum for both methods.

Figure 7 shows the total costs when varying the parameter ρ , which specifies whether or not an edge should be considered between two nodes in the tag set generation algorithm. In our experiments ρ is not an absolute value, as the resemblance values estimated by combsketch and sketch are very different in the absolute sense but they usually hold the same ordering: if a list l_1 has higher true resemblance to l_2 than l_3 this likely holds also

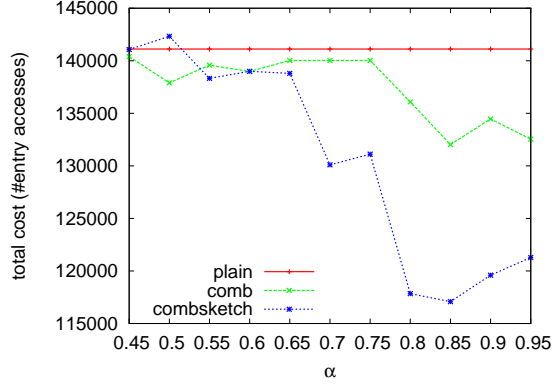


Figure 4: Total_cost values when varying the α parameter. $W=10,000,000ms$, $kdocs=100$, $ctags=75$, $\rho = 0.6$

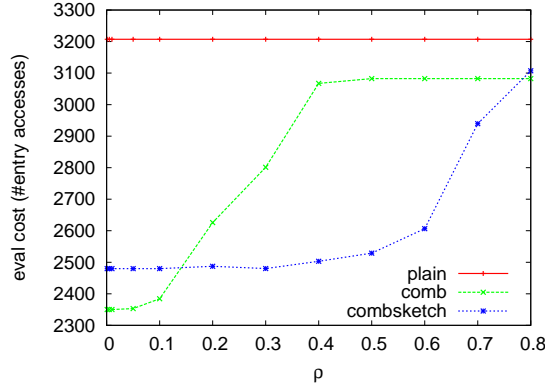


Figure 5: Eval_cost values when varying the ρ parameter. $W=10,000,000ms$, $kdocs=100$, $ctags=75$, $\alpha = 0.8$

in the estimated values by combsketch. So we calculate the highest resemblance value res_{max} and $\rho * res_{max}$ is the threshold considered. We repeat the same procedure for $\rho + step$, each time increasing the resemblance threshold until it reaches 1. This way, we produce smaller tag sets which have high resemblances. So, as observed also in Figure 6 the pre-aggregation cost decreases by increasing ρ . Note that the ρ value where the pre-aggregation cost is actually paid off in evaluations is different for comb and combsketch.

After discovering good parameters for our algorithms, we evaluate our methods by fixing

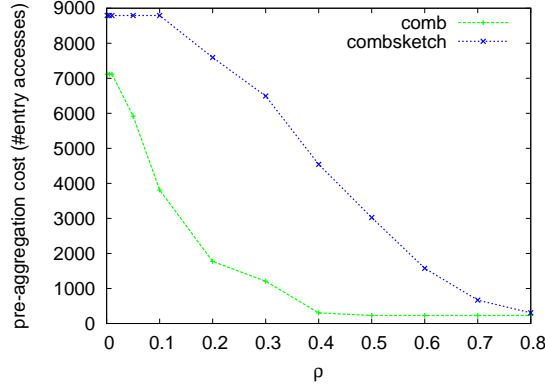


Figure 6: Pre-aggregation_cost when varying the ρ parameter. $W=10,000,000\text{ms}$, $k\text{docs}=100$, $\text{ctags}=75$, $\alpha = 0.8$

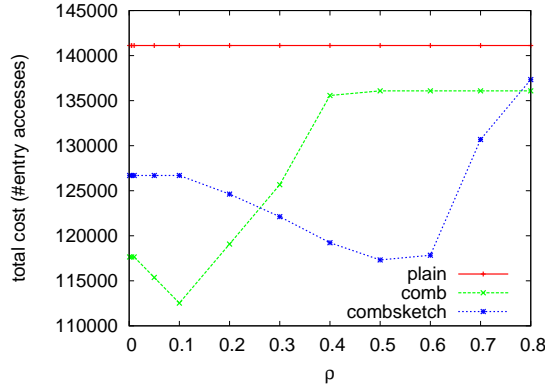


Figure 7: Total_cost when varying the ρ parameter. $W=10,000,000\text{ms}$, $k\text{docs}=100$, $\text{ctags}=75$, $\alpha = 0.8$

those parameters to the best found, and changing the system variables. Figure 8 shows the total cost incurred by the three algorithms when changing the size of the sliding window. Clearly the cost for all three methods increases, as more items are valid at each instance of time, therefore the lists to be accessed are longer. However our algorithms incur much less cost than the plain algorithm. Figure 9 shows the same measure when changing $k\text{docs}$. As expected the TA algorithm can stop earlier for smaller values of $k\text{docs}$. Figure 10 finally, shows the total cost when varying ctags . Since this number defines the number lists we

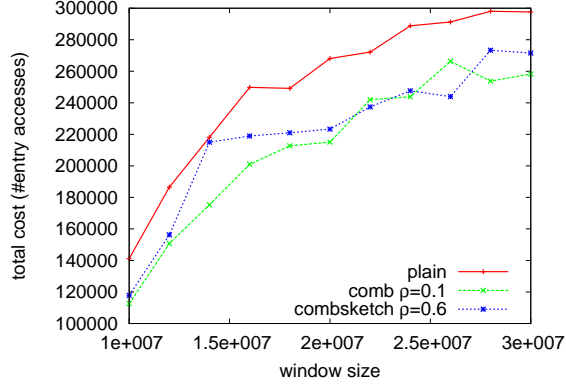


Figure 8: Total_cost when varying the window size, $\alpha = 0.8$, kdocs =100 and ctags =75

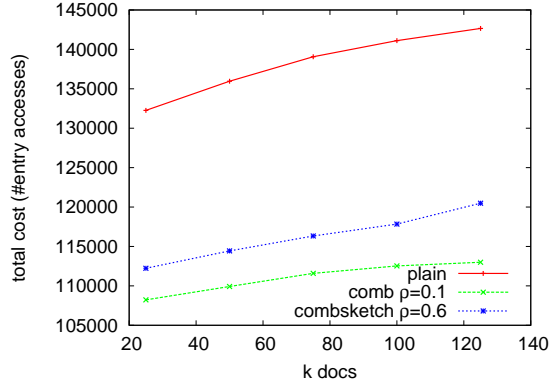


Figure 9: Total_cost when varying kdocs, $\alpha = 0.8$, $W=10,000,000ms$ and ctags=75

should consider in the evaluation, it has a direct effect on total cost. In all three cases, our proposed algorithms incur less cost than the plain method. Although combsketch has only estimates of the true resemblances, its performance gains is very close to comb which has the true resemblance values.

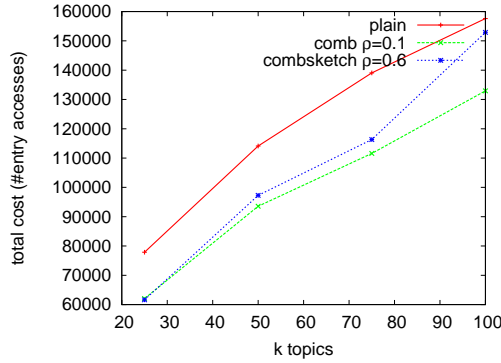


Figure 10: Total_cost when varying ctags, $\alpha = 0.8$, $W=10,000,000ms$ and $kdocs=75$

6 Conclusion

We addressed the problem of continuous monitoring of top- k hottest items over a stream of tagged items such as blog entries or images. We have defined the property of being hot as a top- k aggregation query where the query itself is characterized by the set of most popular tags in a given time period. This causes the top- k query to change over time, hence requires the system to re-evaluate the top- k query from scratch. Our approach is based on the observation that parts of the top- k query are stable for certain time intervals, therefore, do not have to be re-computed in each evaluation phase. As materializing pre-computations of all possible subsets is impractical, we have presented an approximate algorithm to identify the most promising tag subsets (i.e., top- k query ingredients) leveraging FM sketches to predict the suitability of these tag sets. The presented generation method itself gives an easy to use mean to control the amount of pre-aggregated lists.

References

- [ACD⁺98] James Allan, Jaime Carbonell, George Doddington, Jonathan Yamron, and Yiming Yang. Topic Detection and Tracking Pilot Study Final Report, 1998.
- [AGMS02] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking Join and Self-Join Sizes in Limited Storage. *J. Comput. Syst. Sci.*, 64(3):719–747, 2002.
- [APL98] James Allan, Ron Papka, and Victor Lavrenko. On-Line New Event Detection and Tracking. In *SIGIR*, pages 37–45, 1998.
- [BBD⁺02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *PODS*, pages 1–16, 2002.
- [BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

- [CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.
- [CDG07] Toon Calders, Nele Dexters, and Bart Goethals. Mining Frequent Itemsets in a Stream. In *ICDM*, pages 83–92, 2007.
- [CM03] Graham Cormode and S. Muthukrishnan. What’s hot and what’s not: tracking most frequent items dynamically. In *PODS*, pages 296–306, 2003.
- [DGKS07] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Nikos Sarkas. Ad-hoc Top-k Query Answering for Data Streams. In *VLDB*, pages 183–194, 2007.
- [DGKT06] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering Top-k Queries Using Views. In *VLDB*, pages 451–462, 2006.
- [Fag02] Ronald Fagin. Combining Fuzzy Information: an Overview. *SIGMOD Record*, 31(2):109–118, 2002.
- [Fli] Flickr Photo Sharing: <http://www.flickr.com>.
- [FM85] Philippe Flajolet and G. Nigel Martin. Probabilistic Counting Algorithms for Data Base Applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.
- [HCL07] Qi He, Kuiyu Chang, and Ee-Peng Lim. Analyzing feature trajectories for event detection. In *SIGIR*, pages 207–214, 2007.
- [HJSS06] Andreas Hotho, Robert Jäschke, Christoph Schmitz, and Gerd Stumme. Trend Detection in Folksonomies. In *SAMT*, pages 56–70, 2006.
- [HKP01] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD Conference*, pages 259–270, 2001.
- [JY⁺08] Cheqing Jin, Ke Yi, Lei Chen 0002, Jeffrey Xu Yu, and Xuemin Lin. Sliding-window top-k queries on uncertain streams. *PVLDB*, 1(1):301–312, 2008.
- [Kle02] Jon M. Kleinberg. Bursty and hierarchical structure in streams. In *KDD*, pages 91–101, 2002.
- [Kle06] Jon Kleinberg. Temporal dynamics of on-line information streams. In *In Data Stream Management: Processing High-Speed Data*. Springer, 2006.
- [KNRT05] Ravi Kumar, Jasmine Novak, Prabhakar Raghavan, and Andrew Tomkins. On the Bursty Evolution of Blogspace. *World Wide Web*, 8(2):159–178, 2005.
- [KPSV09] Ravi Kumar, Kunal Punera, Torsten Suel, and Sergei Vassilvitskii. Top-k aggregation using intersections of ranked inputs. In *WSDM*, pages 222–231, 2009.
- [MBP06] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, pages 635–646, 2006.
- [MK09] Michael Mathioudakis and Nick Koudas. Efficient identification of starters and followers in social media. In *EDBT*, pages 708–719, 2009.
- [MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.
- [Mut05] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[You] Youtube, Broadcast Yourself: <http://www.youtube.com/>.

[YYY⁺03] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient Maintenance of Materialized Top-k Views. In *ICDE*, pages 189–200, 2003.

Flexible and Efficient Sensor Data Processing – A Hybrid Approach

Claas Busemann, Christian Kuka, Daniela Nicklas, Susanne Boll

OFFIS - Institute for Information Technology
Oldenburg, Germany
busemann|kuka@offis.de
<http://www.offis.de>

Carl von Ossietzky Universität
Oldenburg, Germany
daniela.nicklas|susanne.boll@uni-oldenburg.de

Abstract: The integration of various sensor data into existing software systems is becoming increasingly important for companies and even private users. As the number of embedded devices of all sorts (sensors, mobile phones, cameras etc.) also constantly increases, the development of flexible sensor applications gets more and more difficult. These applications have to handle a large number of sensors transmitting their data in various formats using different protocols. Middleware technologies are a good way to hide the complexity of communication protocols and data processing from the application. However, integrating efficient sensor data processing into a middleware requires several design choices which depend on the planned applications. Usually such systems are either efficient, allowing the processing of large numbers of data streams, or flexible, allowing the easy modification of the processing during runtime. In this paper a hybrid approach is introduced combining the benefits of two popular processing technologies, Service Oriented Architectures (SOA) and Data Stream Management Systems (DSMS), and by that enable the processing of large numbers of data streams while at the same time the system can be flexibly modified. Therefore these two technologies are analyzed to identify the benefits and disadvantages depending on their flexibility and efficiency.

1 Introduction

Today, many low cost and easy to install sensors are available, many more in the future. However, to realize even simple sensor-based applications, the effort is quite high: there are many different interfaces and standards to communicate with the sensor systems. The data read from the sensors is of variable quality and has often to be interpreted, aggregated or otherwise preprocessed before being usable by applications. Since the heterogeneity of sensor systems is so high, applications are bound to certain vendors or proprietary so-called standards. Hence, even if more and more sensor systems are already installed, it is hard to re-use the effort of pre-processing for other applications. Thus, future middleware

architectures for sensor-based applications have to fulfill two major requirements that are often contradicting: flexibility and efficiency. One solution for the flexible orchestration of reusable sensor processing services could be a service-oriented architecture using the Sensor Web Enablement [Ope] framework. Web service technology has many benefits regarding extensibility and standardization. They mostly rely on an event driven processing paradigm, where active services push business events on a service bus and other services subscribe to these basic events to provide higher level information (like complex event processing engines). Finally, the end-user application receives only those events it is really interested in.

When dealing with raw sensor data, this paradigm gets to its limits. Not every sensor measurement is a meaningful event, and the throughput of event buses is not suited for high loads of input data, as we can show in our evaluation. Hence, to close the gap between the high level service oriented world and the low level sensor network world we introduce an hybrid approach: for pre-processing high loads of sensor data, a data stream management system is used which acts as a configurable service within a service oriented architecture. Our evaluation shows that this approach dramatically increases the possible amount of incoming sensor data. The remainder of this paper is organized as follows: Section 2 discusses the related work. In Section 3 two popular sensor data processing technologies are analyzed followed by the description of our hybrid approach in Section 4. Section 5 contains a performance evaluation of the approach. Finally, section 6 draws a resume.

2 Related Work

Sensor data processing has been integrated into several middlewares like the Oracle Sensor Edge Server [Ora10b], the SensWeb platform [Mic08, SNL⁺06], the WISE platform [PHHL04] or the FireEagle platform [Yah07]. These systems use service oriented architectures (SOA) to realize the sensor data processing. This allows the developers to process the incoming sensor data using services and after that provide it to the application. SOA based processing systems are very flexible as they allow the simple integration of application specific services and can easily be orchestrated. User access administration can also be easily integrated, as every service is able to check the access permissions. However, a SOA based system usually cannot handle high-volume streams of incoming sensor data because of the administration overhead that goes along with service oriented architectures.

Other sensor middlewares like the Nexus platform [GBH⁺05], SStreaMWare [GRL⁺08] or the GSN middleware [SA07] are based on Data Stream Management Systems (DSMS). A DSMS is able to process high-volume streams of data in a fast and efficient way. Therefore those systems provide the same extended relational algebra operators like database management systems (DBMS). This algebra is extended through a temporal algebra to perform aggregation and joins on datastreams. In the OpenSource Community there exist a wide range of DSMS implementations like Esper [Esp08] from Espertech, IEP [iep] from the OpenESB community or TelegraphCQ [CCC⁺03]. As there is no public standard for DSMS query languages, all these implementations are using their own query language which are usually based on SQL. However, integrating application specific operators into

DSMS can be extremely complicated as every new operator has not only to be integrated into the DSMS but also into the query language. Adding user access administration to a DSMS can also be very complicated, as most DSMS are not designed to handle user access permissions.

3 Sensor Data Processing

In this section, two popular data processing technologies, Service Oriented Architectures (SOA) and Data Stream Management Systems (DSMS), are analyzed. This is done by evaluating existing open source systems which are based on one of these technologies. As DSMS and SOA based processing technologies are significantly different from each other the evaluation is separated for each of these. Finally, the features of both technologies are compared to each other.

3.1 Service Oriented Architectures (SOA)

SOA based data processing systems are analyzed by identifying the benefits and disadvantages of three popular open source SOA based systems. These are Service Mix 4 [Apa10] developed by Apache Group, OpenESB [Ora10a] developed by Sun and PEtALS [OW210] developed by OW2. As all of these systems are based on a service oriented architecture, they all allow the simple definition of data processing. Therefore, the developer chooses the services she needs to process the incoming sensor data. These can be services executed inside of the SOA but also external services that run on a different server. After that she defines in which order these services are used by linking them logically together. However, there are several features making SOA based systems extremely flexible. As they usually follow the "publish, find, bind" principle using a service broker they are able to discover additional services at runtime. SOA systems usually can be clustered. This makes it possible to run services with high memory or CPU usage on more powerful machines. The deployment of new services without interacting with the running systems (hot deployment) is not supported by some systems. Some systems can be integrated into application servers. This can be very useful when building web based sensor applications. Other systems are able to run as a standalone application which makes it easy to integrate them into ordinary applications. The orchestration of services is often done using BPEL [WCL⁺05] which is a widely accepted standard. Some systems are able to handle access permissions to single services.

However, not all of these features are supported by every SOA implementation. Tab. 1 shows an overview of analyzed systems and the features they support.

The service oriented approach brings a lot of flexibility but it also slows down the processing speed because of the administrative overhead. This is caused by the service broker, which has to decide what service to use next. The use of external services also slows down the system because of the communication overhead. If services are using different

	ServiceMix 4	OpenESB	PEtALS
Clustering	+	+	+
Service Broker	+	+	+
Hot Deployment	+	-	+
Standalone	+	-	+
Embedded	-	+	-
Access Perm.	+	-	+
Orchestration	Camel, EIP	BPEL	BPEL

Table 1: SOA based Systems - Feature Overview

communication protocols, the conversion also increases the processing time. Beyond that, external services can not be controlled as the developer has no influence on their behavior.

3.2 Data Stream Management Systems (DSMS)

Data Stream Management Systems (DSMS) are designed to process high-volume streams of data in a fast and efficient way. Therefore those systems provide the same extended relational algebra operators like database management systems (DBMS) extended through a temporal algebra to perform aggregation and joins on data streams. However, DSMS still differ from each other by features and algebra. In this section three popular open source DSMS are analyzed: TelegraphCQ [CCC⁺03] developed by the Berkeley University, Odysseus [JBG⁺09] developed by the University Oldenburg and Esper [Esp08] developed by EsperTech. As mentioned before, DSMS are usually controlled using a relational algebra. This algebra is translated into a logical operator graph which is deployed by the system. This graphs can be optimized by the system to make the processing more efficient. Some DSMS support the static optimization of operator graphs, which is done before the graph is deployed. Other systems are able to dynamically optimize the graph while the system is running. Clustering is also supported by some of the systems, but the implementations differ from each other. Odysseus for example realizes this by sharing operators on multiple Odysseus instances using a P2P network, while TelegraphCQ technically builds up a shared database which makes it possible to process the operator graph on different instances of the system. DSMS can use different scheduling strategies to optimize the memory and CPU usage. Some systems are able to use graph scheduling, which means that every operator graph runs as an own thread and operators are processed sequentially. Other systems allow operator scheduling, where every operator runs in its own thread. There are also systems that allow hybrid scheduling in which case the single operators or groups of operators run in an own thread. Another important feature is the prioritization of operations inside the system. This can be handled by allowing to prioritize a request to the system or allowing the prioritized processing of selected data by adding a priority to the data tuples. Complex Event Processing (CEP) is used to analyze data streams and search for patterns. As there is no public standard for DSMS, query lan-

guages like SQL for DBMS all the analyzed systems use their own query language which are based on SQL. TelegraphCQ uses the Continues Query Language (CQL), Esper the Event Pattern Language (EPL) and Odysseus SPARQL (for RDF tuple streams) and sSQL (for relational tuple streams). Tab. 1 shows an overview of analyzed systems and the features they support.

	TelegraphCQ	Odysseus	Esper
Static Opti.	-	+	-
Dynamic Opti.	+	-	-
Clustering	+	+	-
G. Scheduling	+	+	+
O. Scheduling	+	+	-
H. Scheduling	-	+	-
Prioritization	-	+	+
CEP	-	+	+
Query Language	CQL	sSQL	EPL

Table 2: DSMS Systems - Feature Overview

The data stream management approach brings a lot of efficiency when filtering and aggregating data but also lacks the possibility to extend the available operators. This is caused by the underlying system which is designed to process data while minimizing memory and CPU usage. As the system has to know each operator at start time to be able to optimize the operator graph and choose the right optimization strategy, the integration of new operators is very complicated.

3.3 Service Oriented Architectures vs. Data Stream Management Systems

The previous sections show that depending on their features SOA based systems and DSMS are primarily different. However, they still share the same application purpose which is processing incoming data using predefined operators. Because of the underlying architecture, SOA systems are extremely flexible. The main benefits of this architecture are the flexible orchestration of services using accepted standards like BPEL, that services can easily be added and modified, that the integration of external services is very easy and that access permissions to services can be handled. The main disadvantage is that the SOA slows down the processing time. Also an optimization of the operator graph is not possible in a way comparable with the optimization inside a DSMS. External operators are easy to integrate but they are also hard to control. This can effect the data processing as they may stop working or change their behavior.

DSMS on the other side are extremely efficient. The main benefits of this architecture are that it is possible to process data in a memory and CPU usage efficient way, that operator graphs can be optimized, that the processing can be optimized using different scheduling strategies, that queries can be prioritized and that complex events can be recognized. The

main disadvantages are that the integration of new operators is difficult, that operators can not be deployed during runtime, that the integration of external operators usually is not possible, that no standard query language exists and that access permissions to operators can not be handled.

However, sensor data processing for applications which have to be flexible and at the same time handle a huge amount of sensor data in a memory and CPU usage efficient way should still be possible. That's why we introduce our hybrid approach, which is explained in detail in the next section.

4 Hybrid Approach

In this section, a hybrid system is introduced which is based on a SOA system and uses a DSMS for efficient processing of sensor data. The basic idea is to reduce the number of data the SOA system has to handle by processing standard operations inside of a DSMS. A pure DSMS can not be used as it can not provide the flexibility that comes along with the SOA system. A schematic view of the system can be found in Fig. 1. To realize this hybrid system, the DSMS is integrated into the SOA system. Therefore a service container is built around the DSMS. The service container is bidirectional connected with a bus system that connects all services in the SOA. This allows the integration of the DSMS into the SOA system as one of its services. To enable the communication between the SOA services and the DSMS operators the sinks and sources of the DSMS are mapped as provider and consumer endpoints for other services of the SOA. A provider endpoint receives sensor data from other services. Each incoming message is transformed into the internal format and pushed into the physical processing plan of the DSMS. Thereby the timestamp of the messages are preserved for usage in timewindow operators. A consumer endpoint publishes the processed sensor data on the bus system for other services. Therefore a new message that includes the processed measurements is generated every time the DSMS produces a new output. This means that other services can not request older values. If they are subscribed to the consumer endpoint they get the newest values from the bus system instead. The service container creates those endpoints when a new sink or source is registered through a processing query and publishes these endpoints for other services. The container is also responsible for the conversation of messages from a provider endpoint to a DSMS source and from a DSMS sink to a consumer endpoint. Sensor data can still be sent directly to the DSMS. Therefore the system allows the communication with the DSMS using a TCP/IP connection. This makes it possible to pre-process data even before the SOA has to handle any of it.

As it now has to be possible to deploy a shared operator graph, which includes operators of the SOA and the DSMS, a framework is built around the system which allows the description, storage and deployment of shared operator graphs.

The description of the shared operator graphs is realized using XML. Operators are always based on operator types which are predefined by the system. One of these operator types is the "DSMS operator". Every other operator type represents an operator of the SOA. When

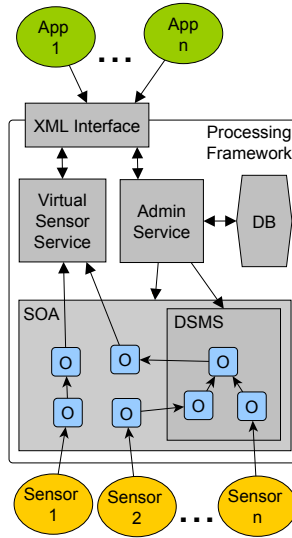


Figure 1: Schematic View of the Hybrid System

creating an operator based on an operator type using the XML interface the operator is first saved in a database. The system also allows the creation of inputs and outputs. Inputs have to define the ID of a sensor data stream and are used to transmit data into the processing framework. Outputs have to define a unique name and are used to receive processed data out of the processing framework. The user then has to orchestrate the inputs, outputs and operators by linking them together. After all operators and links are saved in the database, the user can use the XML interface to deploy the operator graph. The “Admin Service” of the processing framework then accesses the database and creates the orchestration for the SOA system and the DSMS. If the user decides to change the operator graph she can always change its database representation. After that the graph has to be redeployed. After the operator graph is deployed, the incoming sensor data is received by the SOA or the DSMS. The incoming data then is processed by the SOA and DSMS operators depending on the deployed orchestration. After the sensor data is processed it can be accessed by the application using the “Virtual Sensor Service”.

5 Performance Evaluation

In this section a SOA based system, a DSMS and the hybrid approach are compared using a performance evaluation. As the evaluation implementation of the hybrid approach is based on Esper and ServiceMix 4 these systems are also used as stand alone systems for the evaluation.

5.1 Evaluation scenario

The evaluation scenario is based on a community web application that allows the users to monitor the position of their pet.

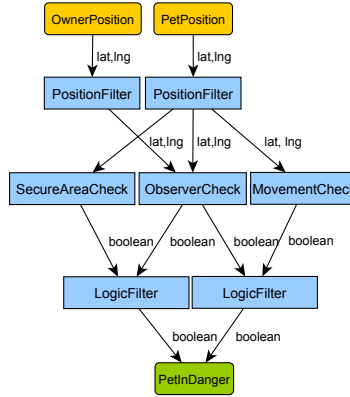
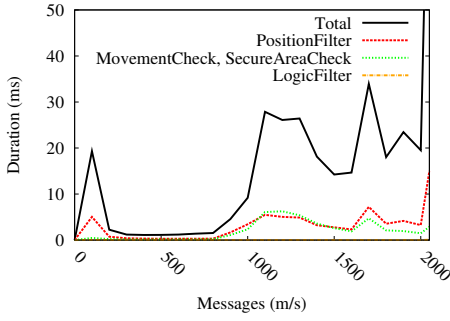


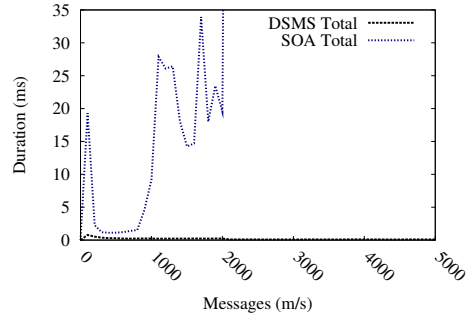
Figure 2: Schematic View of Petfinder Processing

Fig. 2 shows a schematic view of the processing for the petfinder application. Position information is transmitted from the pet and the owner. The position filter (PositionFilter) is used to filter invalid data. After that the valid position data is transmitted to operators which check if the pet is in danger. They check if the pet is in a secure area (SecureAreaCheck), if it is observed by its owner (ObserverCheck) and if it has moved over the last time (MovementCheck). After that, these operators transmit a boolean value that indicates if the pet may be in danger or not to logic filters (LogicFilter). The logic filters make sure that no alarm is sent while the pet is observed. They also check if the events occurred at the same time. If the “PetInDanger” operator receives “true” from one of these filters the pet is in danger and a alarm message has to be sent.

The evaluation is done using a worst case scenario. Pets are never observed, never in a secure area and do not move. This forces the processing system to handle a huge amount of sensor data as every incoming sensor value has to be processed by every operator. It starts with 100 owners monitoring 100 pets. Owners and pets send a new position every five seconds. All data is received during one second giving the system 4 seconds to process the data before new data is transmitted. The number of owners and pets is increased by 100 every 50 seconds. 50% of the incoming pet and owner positions are illegal and have to be filtered. The performance of each system is detected by measuring the average time between transmitting and receiving a single data value. The test system consist of a Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz running Linux Debian with 2.6.32-3-amd64 Kernel and 4GB RAM. The Java(TM) SE Runtime Environment version 1.6.0.20 is used.



(a) Performance Evaluation of ServiceMix 4



(b) Performance Evaluation of Esper

Figure 3: Performance Evaluation of Single Systems

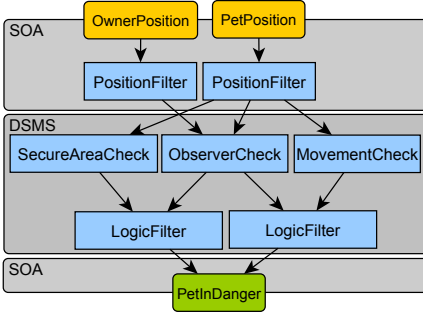
5.2 ServiceMix 4

ServiceMix 4 is tested using the release version (4.0.0). As ServiceMix 4 does not have any operators needed for the application scenario the operators are self implemented. Because ServiceMix provides the developer with a simple interface structure when programming operators and the integration is as simple as putting a jar file into a folder, the necessary operators can be programmed and integrated within one day.

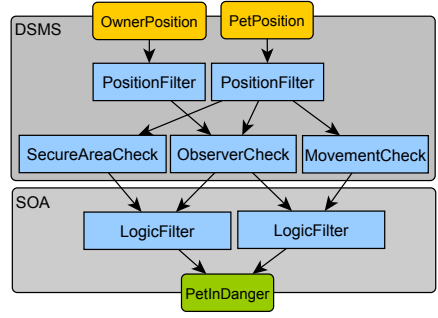
The results of the performance evaluation can be found in Fig. 3a. The peak at the beginning is a result of the binding procedure of the services when the first message is processed. But it shows that ServiceMix 4 is easily able to handle about 700 incoming data streams within 1 seconds processing time. At that point every sensor value is processed in about 2ms. As soon as the number of data streams comes near to 1.000, the systems is no longer able to handle them without extremely slowing down. The system now needs about 30ms to handle one data stream. When more than 2.000 data streams are processed, ServiceMix 4 is no longer able to handle the amount of data and stops processing. This is caused by the bus of ServiceMix which is only able to handle a fixed number of data streams. Fig. 3a also shows the processing time of each operator. However, the evaluation shows that most of the processing time is needed for the communication between the operators, as the sum of the processing time of all operator is often about 10 ms below the total processing time.

5.3 Esper

Esper is evaluated using the release version 2.2.0. All needed operators are already integrated into Esper. The orchestration is done by writing a EPL request. The results of the performance evaluation can be found in Fig. 3b. It shows that Esper can easily handle up to 5.000 in 1 seconds processing time. Every sensor value is processed in about 1ms what is far below the time that ServiceMix 4 needed. Because of the closed system approach of



(a) Evaluation Scenario 1



(b) Evaluation Scenario 2

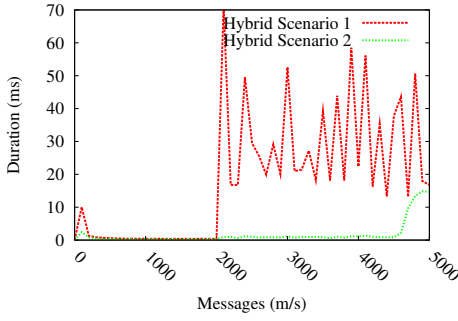
Figure 4: Evaluation Scenarios

Esper it is not possible to monitor the processing time of a single operator without spoiling the evaluation results. However, as the implementation of the ServiceMix operators does not differ too much from the implementation of the Esper operators it should be clear that the time benefit is achieved by the missing communication overhead.

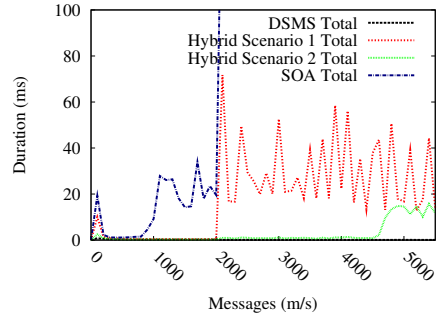
5.4 Hybrid Approach

The hybrid approach is tested using the basic architecture described in section 4. Therefore the operator graph is split into parts. These parts are either processed by the ServiceMix 4 or Esper. The communication between these parts is handled by the ServiceMix 4. Two different shared operator graphs are evaluated. The first one can be found in Fig. 4a.

In this evaluation scenario, the “SecureAreaCheck” operator, “ObserverCheck” operator, “MovementCheck” operator and the two “LogicFilter” operators are handled by the DSMS. The processing of these operators should be much faster than the rest, because of the missing administrative overhead. The PositionFilter operators are still part of the SOA system and can therefore easily be changed or replaced during runtime. The second evaluation scenario can be found in Fig. 4b. The SOA is only used for the “LogicFilter” operators. Everything else is handled by the DSMS. In this scenario the SOA only has to handle a minimum of data streams as the DSMS already reduces their number using the “PositionFilter” operators. The results of both evaluations can be found in Fig. 5a. The result of the first evaluation scenario shows, that the processing of a part of the operator graph inside a DSMS reduces the average processing time of every data stream. However, the processing still gets pretty slow when more than 2.000 data streams are processed. This is exactly the point where ServiceMix 4 stopped working in the prior evaluation. The second evaluation scenario shows that even more data streams can be processed if the DSMS is used to filter data streams, which do not have to be processed by the SOA. As the “PositionFilter” operators which reduce the amount of data streams by 50% are now



(a) Performance Evaluations of Hybrid System



(b) Performance Evaluation of All Systems

Figure 5: Performance Evaluations

processed by Esper, ServiceMix only has to process a minimum of data streams. This reduces the communication overhead and allows the complete system to process about 4,500 data streams before slowing down. The average processing time is also extremely low at about 2 ms. The results of all tests can be found in Fig. 5b. The evaluation proves that using a hybrid system, which combines the benefits of a SOA system and a DSMS, is a good way to build a flexible and efficient data processing system. The hybrid system caused a drastically performance rise in both evaluation scenarios in comparison to SOA system. However, a pure DSMS seems to be the fastest system, but using the hybrid system operators can still be implemented inside of the SOA allowing them to benefit from the flexibility that comes along with those systems. At the same time selected operators can be integrated into a DSMS to reduce the communication overhead and increases the efficiency of the processing.

6 Conclusion

In this paper two popular data processing technologies, Service Oriented Architectures (SOA) and Data Stream Management Systems (DSMS), are analyzed. Their benefits and disadvantages have been identified and analyzed. Based on this evaluation a hybrid system has been developed that increases the performance of a SOA system by integrating a DSMS into it. As the DSMS does not have to be used for all operations the hybrid system still provides all the flexibility that comes along with the SOA. The positive effects of this hybrid approach are proven by a performance evaluation.

References

[Apa10] Apache Software Foundation. ServiceMix 4. <http://servicemix.apache.org/>

home.html, 2010.

- [CCC⁺03] Sirish Chandrasekaran, Sirish Ch, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World, 2003.
- [Esp08] Espertech Inc. EsperTech: Event Stream Intelligence. <http://camel.apache.org/manual/>, 2008.
- [GBH⁺05] M. Grossmann, M. Bauer, N. Honle, U. Kappeler, D. Nicklas, and T. Schwarz. Efficiently Managing Context Information for Large-Scale Scenarios. In *Proceedings of Pervasive Computing and Communications, IEEE Computer Society*, 2005.
- [GRL⁺08] Levent Gurgun, Claudia Runcancio, Cyril Labbé, André Bottaro, and Vincent Olive. SStreamWare: a service oriented middleware for heterogeneous sensor data management. In *ICPS '08: Proceedings of the 5th international conference on Pervasive services*, pages 121–130, New York, NY, USA, 2008. ACM.
- [iep] IEP - Open Source Complex Event Processing (CEP) and Event Stream Processing (ESP) Engine. <https://open-esb.dev.java.net/IEPSE.html>.
- [JBG⁺09] Jonas Jacobi, André Bolles, Marco Grawunder, Daniela Nicklas, and H.-Jürgen Appelrath. A physical operator algebra for prioritized elements in data streams. *Computer Science - Research and Development*, 2009.
- [Mic08] Microsoft Research. SenseWeb Project. Technical report, Microsoft Research, Redmond, USA, 2008.
- [Ope] Open Geospatial Consortium (OGC). Sensor Web Enablement (SWE). Specification.
- [Oral0a] Oracle Corporation. OpenESB. <https://open-esb.dev.java.net>, 2010.
- [Oral0b] Oracle Corporation. Oracle Sensor Edge Server. http://www.oracle.com/technology/products/sensor_edge_server/index.html, 2010.
- [OW210] OW2 Consortium. PEtALS. <http://petals.ow2.org>, 2010.
- [PHHL04] Rui Peng, Kien A. Hua, and Georgiana L. Hamza-Lup. A Web Services Environment for Internet-Scale Sensor Computing. In *SCC '04: Proceedings of the 2004 IEEE International Conference on Services Computing*, pages 101–108, Washington, DC, USA, 2004. IEEE Computer Society.
- [SA07] Ali Salehi and Karl Aberer. GSN, Quick and Simple Sensor Network Deployment. In *European conference on Wireless Sensor Networks*, 2007.
- [SNL⁺06] Andre Santanche, Suman Nath, Jie Liu, Bodhi Priyantha, and Feng Zhao. SenseWeb: Browsing the Physical World in Real Time. In *Demo Abstract, ACM/IEEE IPSN 2006, Nashville, TN*, April 2006.
- [WCL⁺05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. 2005.
- [Yah07] Yahoo. FireEagle: Centralized management of user location. Technical report, Yahoo Research, 2007.

Feature-Based Graph Similarity with Co-Occurrence Histograms and the Earth Mover's Distance

Marc Wichterich, Anca Maria Ivanescu, Thomas Seidl
Data Management and Exploration, RWTH Aachen University
{wichterich, ivanescu, seidl}@cs.rwth-aachen.de

Abstract: Graph structures are utilized to represent a wide range of objects including naturally graph-like objects such as molecules and derived graph structures such as connectivity graphs for region-based image retrieval. This paper proposes to extend the applicability of the Earth Mover's Distance [RTG98] (EMD) to graph objects by deriving a similarity model with a representation of structural graph features that is compatible with the feature signatures of the EMD. The aim is to support the search for a graph in a database from which the query graph may have originated through limited structural modification. Such query graphs with missing or additional vertices or edges may be the result of natural processes of decay or mutation or may stem from measuring methods that are inherently error-prone, to name a few examples.

1 Introduction and Related Work

Graphs are widely used data structures for modeling complex objects. For example, in computer vision and pattern recognition graphs are extracted from complex objects, stored in databases, and are used for graph-based shape recognition [SKK04] or for object recognition in general [HHEW04]. In the biomedical field, Takahashi investigates the structural similarity of chemicals with similar biological activity by using graphs to represent the structure of the chemicals [Tak04]. These applications exemplify the need for graph similarity measures that allow for the clustering of graphs in a database, or for finding graphs in a graph database that are similar to a query graph.

In the absence of a canonical representation of graphs, deciding if two graphs are isomorph (i.e., identical but for a renaming of the vertices) is a computationally expensive task. Its generalization, the subgraph isomorphism problem, is known to be NP-complete. When attempting to find all graphs in a database that contain a subgraph that is isomorph to some query graph, it is possible to use lower-bounding filtering techniques to quickly rule out some candidates and refine the rest with the computationally expensive exact matching. For example, the GraphGrep approach indexes labels along paths within a graph to perform the filtering [SWG02].

For similarity search, deciding whether (sub)graphs are isomorph does not suffice. In the case of the two graphs not being identical, similarity search requires an assessment of the degree to which the graphs in question differ from another such that graphs in a database can be sorted by similarity to a query graph.

The comparison of two graphs can be performed by directly considering the structure of the graphs. This approach is, for example, taken by the graph edit distance [BA83] that calculates how many transformations have to be performed to turn one graph into the other, and also by measures that consider common subgraphs or the size of the largest common subgraph [BS98]. While these measures are suitable for small graphs and for graphs with limitations regarding their structure and/or the operations that may be performed (e.g., the degree-2 edit distance for connected, undirected, acyclic trees [ZWS96]), even medium-sized general graphs quickly lead to a query processing time that is bound to overburden the patience of the user.

Akin to content-based image retrieval, feature-based graph similarity models instead derive (approximate) structural information from the graphs and assess the similarity of the graphs based on these features. For example, so-called spectral approaches [Ume88, LWH03] compare graphs based on an eigen-decomposition of the adjacency matrix. The model presented in [PM99] compares two graphs by computing the difference in the number of nodes that have a given connectivity degree. The latter is the basis for the generalized approach described in this short paper. We collected connectivity information along paths in graphs and represent the information in a way that allows graphs to be flexibly compared using the EMD. In a recent related approach, graphs derived from images have been compared using the EMD [GXTL08]. However, the approach uses the EMD to compare the direction of edges/lines that occur in the graph and thus requires the vertices to have a spatial position. The approach described here is devised in a more general way as it does not make such an assumption.

2 Preliminaries

The basic graph-related definitions for concepts used in the rest of the paper are given in this section.

A general graph with at most one edge from one vertex to another is defined via its set of vertices and its edge relation.

Definition 1 (*Graph*)

A graph G of size m is a tuple $G = (V, E)$ with vertices $V = \{v_1, \dots, v_m\}$ and edges $E \subseteq V \times V$.

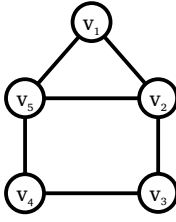
If a graph does not have single-vertex loops (i.e., the edge relation is irreflexive) and is undirected (i.e., the edge relation is symmetric), it is called simple.

Definition 2 (*Simple Graph*)

Given a graph $G = (V, E)$, G is simple iff for all $v, w \in V$

$$(v, w) \in E \Leftrightarrow (w, v) \in E \text{ and } (v, w) \in E \Rightarrow v \neq w.$$

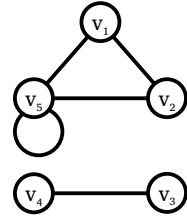
All graphs examined in the remainder of this paper are assumed to be simple.



(a) Graph G_1

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	0	0	1
v_2	1	0	1	0	1
v_3	0	1	0	1	0
v_4	0	0	1	0	1
v_5	1	1	0	1	0

(b) Adjacency matrix of G_1



(c) Graph G_2

Figure 1: Example graphs

If all vertices of a graph are connected to all other vertices of the graph by a series of edges, it is called connected.

Definition 3 (Connected Graph)

Given a graph $G = (V, E)$, G is connected iff for all $v, w \in V$:

$$(v \neq w) \Rightarrow (v, w) \in E$$

$$\vee (\exists v_{i_1}, \dots, v_{i_{m'}} \in V : (v, v_{i_1}) \in E \wedge (v_{i_{m'}}, w) \in E$$

$$\wedge \forall 1 \leq j \leq m' - 1 : (v_{i_j}, v_{i_{j+1}}) \in E).$$

The graphs in the database are assumed to be connected in this paper. A query graph with missing vertices or edges may however break down into several non-connected components.

The degree of a vertex in a graph is the number of other vertices it is directly connected to.

Definition 4 (Vertex Degree Function)

Given a graph $G = (V, E)$, the outgoing vertex degree function $\delta^G : V \rightarrow \mathbb{N}_0$ for G is defined by

$$\delta^G(v) = |\{w \in V | (v, w) \in E\}|.$$

The ingoing vertex degree function can be defined analogously. For the simple graphs of this paper, the two functions are identical and thus do not have to be differentiated here.

The example graph G_1 in Figure 1(a) is a simple, connected graph with 5 vertices and 6 edges. Figure 1(b) gives the adjacency matrix of G_1 where an entry of 1 indicates the existence of an edge while an entry of 0 indicates the absence of an edge between two vertices. As a result of Definition 2, the diagonal entries are all zero and the matrix is symmetric. The degree of a vertex equals the row sum in the adjacency matrix. Vertices v_1 , v_3 , and v_4 have degree 2 while vertices v_2 and v_5 have degree 3. The graph G_2 is neither simple (due to the loop at v_5) nor connected (due to having two separate components).

3 Graph Similarity Model

In order to find graphs in a database that might be related to a query graph through a process of decay, mutation or generally structural change, a representation of statistical graph features is proposed in Section 3.1 and distance measures suitable for the feature representation are given in Section 3.2. The similarity of two graphs can be assessed by combining these two parts.

3.1 Graph Feature: Degree Co-Occurrence Multisets

A representation of graph features that encodes structural information is required for detecting small structural changes between graphs in a feature-based approach. In this section, statistical features of the vertices that occur in the graphs and their connectivity relationship are discussed. In the simplest form, a graph can be represented by the distribution of the degrees of its vertices as in [PM99]. However, by looking at each vertex separately, one of the core concepts of graphs is ignored. Graphs are useful as they model relationship information between the vertices. Thus, this section proposes to utilize statistical information on the co-occurrence of vertices. In this way, the feature representation encodes which kinds of vertices are connected within a graph – and how frequent this coupling occurs. The co-occurrence concept can be generalized by looking at occurrences along paths in the graph and noting which kinds of vertices occur close to each other / in sequence. In the following definitions, the generalized co-occurrence concept is formally introduced on the basis of vertex degrees as this information is common to all graphs. If other categorical information (e.g., vertex class labels) is available, the approach could be adapted to incorporate that information.

Definition 5 (Simple Vertex Path)

With $G = (V, E)$ as a graph, the $(m + 1)$ -tuple $(v_{i_0}, \dots, v_{i_m}) \in V^{m+1}$ is a simple (non-looping) vertex path of length m in G iff

$$\forall 0 \leq j < j' \leq m : v_{i_j} \neq v_{i_{j'}}$$

and

$$\forall 0 \leq j < m : (v_{i_j}, v_{i_{j+1}}) \in E.$$

The set of all simple paths of length m in G is denoted as P_m^G .

For the cases of path lengths $m = 0$ and $m = 1$, sets P_0^G and P_1^G equal the set of vertices V and the set of edges E . Using the set of simple paths of length m , a co-occurrence multiset of degree m captures the frequencies of vertex class (here, vertex degree) sequences.

Definition 6 (Vertex Degree Co-occurrence Multisets)

With $G = (V, E)$ as a graph, the Vertex Degree Co-Occurrence Multiset D_m^G of degree m for graph G is defined as a tuple

$$D_m^G = (DS_m^G, f_m^G)$$

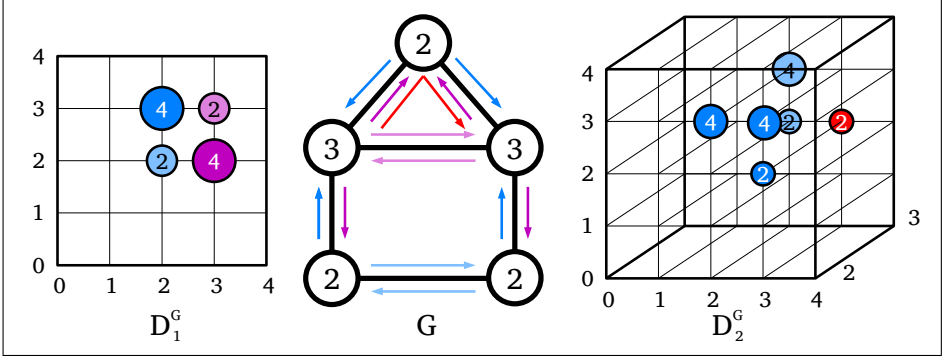


Figure 2: Visualization for the multiset feature representation of a graph G

where

$$DS_m^G = \{(\delta^G(v_{i_0}), \dots, \delta^G(v_{i_m})) \mid (v_{i_0}, \dots, v_{i_m}) \in P_m^G\}$$

is the set of all vertex degree sequences occurring on paths of length m in G and

$$f_m^G(dg_0, \dots, dg_m) = |\{(v_{i_0}, \dots, v_{i_m}) \in P_m^G \mid \forall 0 \leq j \leq m : dg_j = \delta^G(v_{i_j})\}|$$

the frequency function of such sequences in G .

The set DS_m^G abstracts from individual vertices by only considering their type (i.e., vertex degree in this example). The degree m of the multiset is not related to the degree of the vertices in the graphs but only to the length of the examined paths.

As an example, the graph G_1 in Figure 1(a) has five paths of length $m = 0$ (i.e., $P_0^{G_1} = V = \{v_1, \dots, v_5\}$). The occurring vertex degrees are $DS_0^{G_1} = \{(2), (3)\}$ with frequencies $f_0^{G_1}(2) = 3$ and $f_0^{G_1}(3) = 2$. For $m = 1$, there are twelve paths (i.e., two per edge). The combinations of vertex degrees occurring along those paths are $DS_1^{G_1} = \{(2, 2), (2, 3), (3, 2), (3, 3)\}$. The frequencies of those paths are $f_1^{G_1}(2, 2) = 2$, $f_1^{G_1}(2, 3) = 4$, $f_1^{G_1}(3, 2) = 4$, and $f_1^{G_1}(3, 3) = 2$. The set of vertex degree sequences $DS_2^{G_1}$ is of cardinality 6 and $DS_3^{G_1}$ of cardinality 8. The experiments in Section 4 show good results for m as low as 2. For greater lengths, techniques such as random path sampling could be applied to speed up the feature extraction process.

Figure 2 shows a visualization of the co-occurrence multisets D_1^G (on the far left) and D_2^G (on the far right) in the form of bubble charts. The x , y , and z axes denote the degree of the first, second, and third vertex on a path in G . The size of the bubbles is proportional to the frequency of the according vertex degree sequences that is also denoted inside the bubble. For $m = 1$, the short arrows next to the graph in the middle of the figure show all paths (i.e., edges) that contribute to the multiset D_1^G . The long arrow in the upper section of the graph shows a path that contributes to the bubble at coordinate 3-2-3 in the far right of the figure representing D_2^G .

3.2 Similarity Measure

With the above definitions, a co-occurrence multiset can be associated with each graph in the database and with the query graph. Graph similarity can then be assessed in terms of co-occurrence multisets that capture statistical information on the structure of the graphs. Next, we describe how this feature representation can be compared via distance measures.

3.2.1 Element-Wise Multiset Comparison

A first approach is to treat the multisets as sparse representations of high-dimensional vectors. Since the multisets are finite, norm-based distance measures such as the L_p distances can be adapted to compare two graphs represented by such multisets.

Definition 7 (L_p Distance on Vertex Degree Co-Occurrence Multisets)

Given two graphs G_1 and G_2 with associated vertex degree co-occurrence multisets $D_m^{G_1} = (DS_m^{G_1}, f_m^{G_1})$ and $D_m^{G_2} = (DS_m^{G_2}, f_m^{G_2})$ according to Definition 6, the L_p distance between the two multisets is defined as

$$d_{L_p}(D_m^{G_1}, D_m^{G_2}) = \left(\sum_{ds \in (DS_m^{G_1} \cap DS_m^{G_2})} |f_m^{G_1}(ds) - f_m^{G_2}(ds)|^p + \sum_{ds \in (DS_m^{G_1} - DS_m^{G_2})} |f_m^{G_1}(ds)|^p + \sum_{ds \in (DS_m^{G_2} - DS_m^{G_1})} |f_m^{G_2}(ds)|^p \right)^{1/p}.$$

In the case of $m = 0$ and $p = 1$, the similarity model reflects the one of [PM99] where graphs are compared using simple vertex degree histograms and the Manhattan distance.

3.2.2 Transformation-Based Multiset Comparison

Another possibility is to employ similarity measures that can inherently cope with weighted feature sets instead of just feature vectors such as the EMD [RTG98]. For this purpose, we first introduce the feature signatures used as an input for the EMD, followed by the definition of the EMD.

Definition 8 (Feature Signatures)

Given an object o represented by features f_1, \dots, f_k in a feature space FS , and an n -clustering C_1, \dots, C_n of these features, the feature signature s^o of the object o is defined as a finite set of tuples from $FS \times \mathbb{R}$:

$$s^o = \{(r_1^o, w_1^o), \dots, (r_n^o, w_n^o)\}$$

where $r_i^o \in FS$ represents the feature cluster C_i and $w_i^o = \frac{|C_i|}{k}$ is the relative cardinality or weight/mass of the according cluster.

The EMD itself is defined as a linear optimization problem. The similarity between two signatures s^o and s^q is defined as the minimal cost for transforming the signature s^o into the signature s^q where a ground distance gd determines the cost of transforming/moving a unit of mass from a cluster of the first signature to a cluster of the second signature. Linear constraints on the movement of mass describe the set of feasible combinations of transformations.

Definition 9 (Earth Mover's Distance (EMD))

Given two signatures s^o , s^q , and a ground distance gd , the EMD between s^o and s^q is defined as the minimum over feasible transformations $F \in \mathbb{R}^{|s^o| \times |s^q|}$:

$$EMD_{gd}(s^o, s^q) = \min_F \left\{ \frac{1}{\tilde{w}} \sum_i \sum_j F[i, j] \cdot gd(r_i^q, r_i^o) \right\}$$

under linear constraints

$$\begin{aligned} \forall i, j : F[i, j] &\geq 0 \\ \forall i : \sum_j F[i, j] &\leq w_i^q \\ \forall j : \sum_i F[i, j] &\leq w_j^o \\ \sum_i \sum_j F[i, j] &= \tilde{w} \end{aligned}$$

with $\tilde{w} = \min\{\sum_{i=1}^n w_i^o, \sum_{i=1}^m w_i^q\}$.

Intuitively, the first group of constraints ensures that earth is only moved from clusters of s^o to clusters of s^q , the second and third group of constraints ensures that no more mass is removed from or moved to the clusters than their respective weight permits and the last constraint ensures that in total as much mass as possible is moved.

The similarity of two graphs can be assessed using the EMD by defining a transformation from the multisets to the signatures of the EMD. The co-occurrence multisets are a close match to the signatures that the EMD takes as its input.

Definition 10 (Feature Signatures of Graphs)

Given a graph G with an associated vertex degree co-occurrence multiset $D_m^G = (DS_m^G, f_m^G)$, the feature signature s_m^G of G for comparison with the Earth Mover's Distance is defined as

$$s_m^G = \left\{ (r, w) \mid r \in DS_m^G \wedge w = \frac{f_m^G(r)}{|DS_m^G|} \right\}.$$

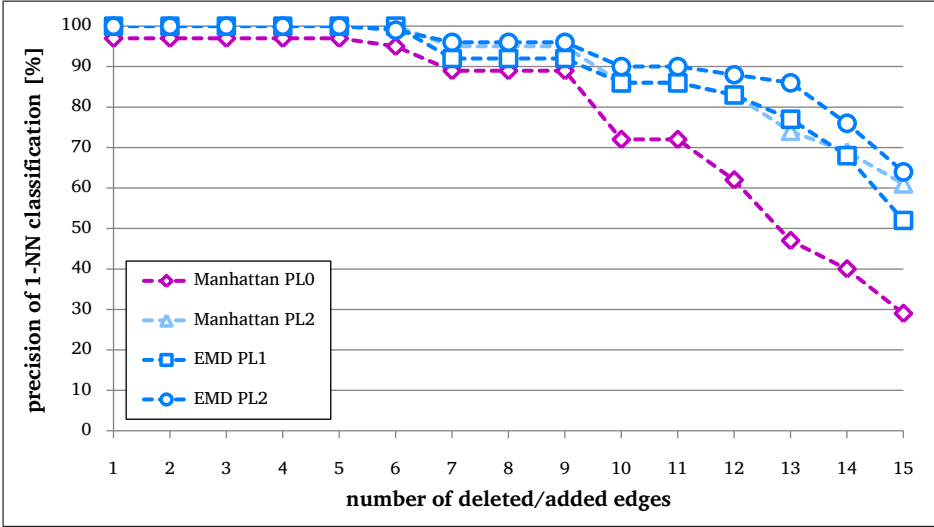


Figure 3: Adding/deleting edges at random; $|DB| = 1000$

The cost for transforming one degree sequence into another one can be defined via a ground distance function. In the simplest case, the sequences can be treated as vectors from \mathbb{N}_0^m and compared using Minkowski distance measures d_{L_p} . In this way, a degree sequence that deviates from another for example by starting with a degree of 3 instead of 4 will induce a lower transformation cost than one that starts with 1 instead of 4. Distance measures such as the Edit Distance, which take the sequential character of the representatives r into account, could also be employed. For undirected graphs, the fact that each sequence of vertex degrees appears twice in both directions should be accounted for by adjusting either the signature definition or the ground distance.

4 Preliminary Experimental Results

For the preliminary experiments shown here, a number of synthetic graph databases of differing cardinalities were created using the method detailed in [VL05] based on sequences of vertex degrees following a power-law distribution with modifications to ensure that the graphs are connected and simple. All graphs randomly generated in this fashion had 100 vertices and 150 edges. The average vertex degree was set to 3, resulting in power-law graphs with a relatively large number of low degree vertices and a relatively low number of high degree vertices.

In the first set of experiments, 100 graphs were randomly chosen from the database as the basis for 15 query graphs each that represent different levels of structural deviation regarding the edge relation. For each of the 15 levels, a random edge was either inserted or deleted with equal probability. Not accounting for edges that may have been deleted

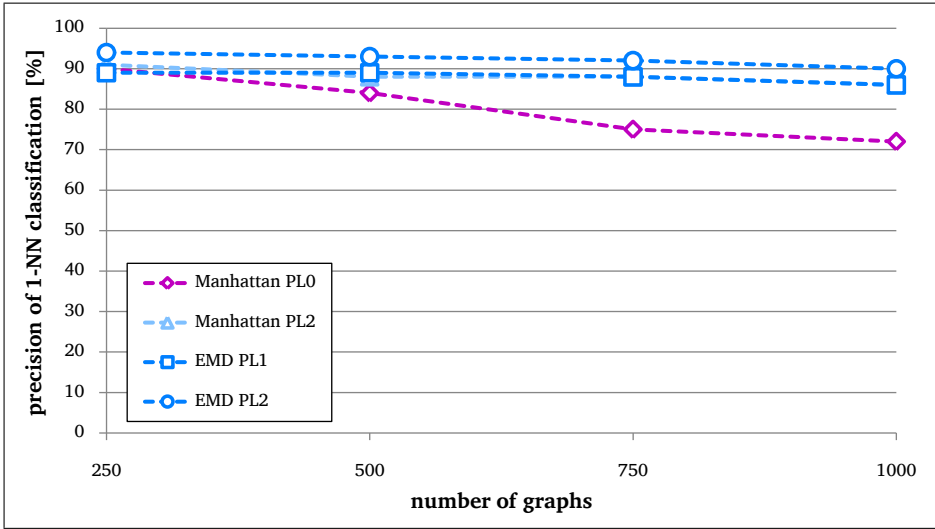


Figure 4: Adding/deleting edges at random; 10 edges added/deleted

and consecutively been added again, up to 10% of the the edge relation may have been changed in this process.

The vertical axis of Figure 4 shows how often the graph on which the query graph was based was identified as the most similar one out of all 1000 graphs in the database. A greater path length (denoted as PL in the figure) for the vertex degree co-occurrence multisets results in a similarity model that is more robust with regard to the structural change for this experiment. The greater the structural difference, the more can the multisets based on longer paths distinguish themselves from those of lower degree. The Manhattan distance on simple vertex degree histograms (cf. [PM99]) is always outperformed by the multisets of higher degree (i.e., based on longer paths) in this experiment. The EMD with a Manhattan ground distance slightly outperforms the Manhattan distance for equal path lengths. The EMD for path length zero is not plotted here, as the results equal those of the Manhattan distance in the case of a one-dimensional feature space and Manhattan ground distance.

Figure 4 shows that the higher degree multisets are also less influenced by the cardinality of the database. Even though the database size on the right is four times the size of the database on the left, the number of times that the original graph from the database is not identified as the most similar one to the query graph only slightly increases from 9 out of 100 to 14 out of 100 for the EMD with path length two. The degree histogram approach jumps from 10 out of 100 to 28 out of 100 for the same increase in database size.

The two figures 4 and 4 show the results of according experiments when considering structural change that is not limited to the edge relation. Instead, random vertices were removed together with their adjacent edges. As is to be expected due to the greater level of structural change, all approaches show a faster decrease of the precision with which they can

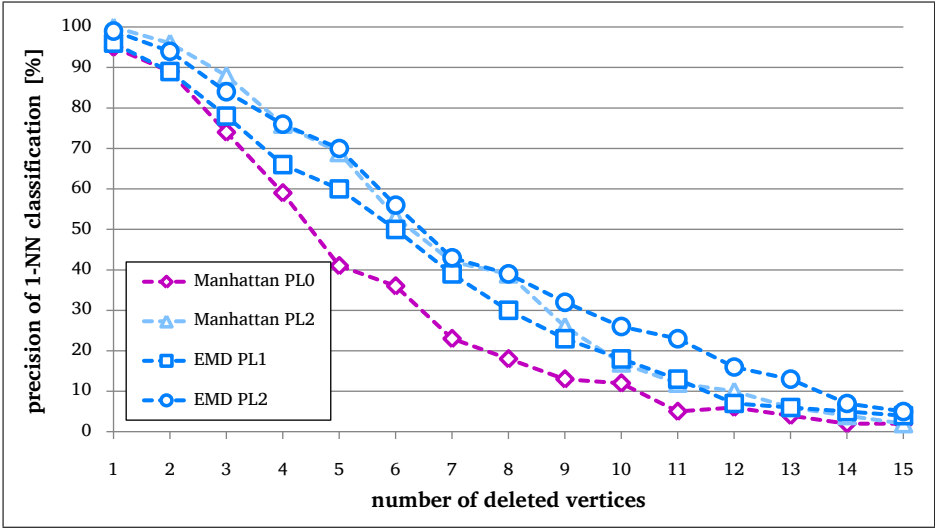


Figure 5: Deleting vertices at random; $|DB| = 1000$

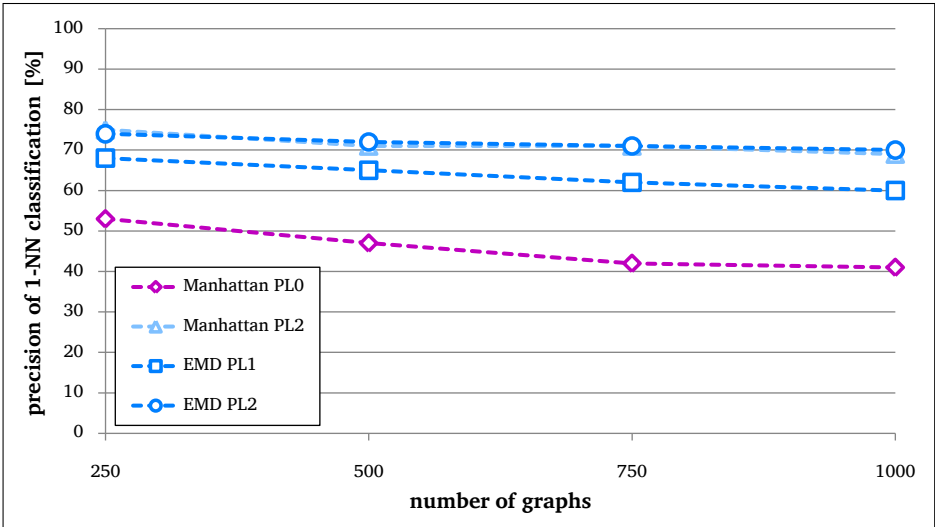


Figure 6: Deleting vertices at random; 5 deleted vertices

identify the original graph in the database. Greater path lengths still produced better results in these experiments while the EMD with a Manhattan ground distance was only able to outperform the normal Manhattan distance for more severe levels of structural change in this case.

5 Conclusion and Outlook

In this short paper, we showed how complex data objects in the form of graphs can be compared using the EMD by defining a suitable representation of graph features that capture statistical information regarding the structure of the graphs. In this way, it is possible to identify graphs that resulted from some other graph through a process of structural change without having to resort to typically very expensive similarity measures that directly take the graph structure into account.

The general viability of the approach was shown using a Manhattan ground distance for the EMD together with vertex degrees as the sole information regarding the vertices. For this ground distance a projection-based lower bound for the EMD [CG97] can be applied in a filter step in order to gain efficiency, especially for higher degrees of the multiset. Also the EMD-L1 algorithm from [LO07] can be employed to speed up retrieval. While the preliminary results using this simple ground distance were generally good, the Manhattan ground distance potentially limits the benefits of longer co-occurrence sequences that are used as signature component representatives for the EMD. Other ground distances that take the sequence character of the feature representatives (i.e., sequences of vertex degrees in this case) into account may present an opportunity to further improve the technique.

6 Acknowledgments

This work is partially funded by DFG grant SE 1039/1-3.

References

- [BA83] Horst Bunke and Gudrun Allermann. Inexact Graph Matching for Structural Pattern Recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.
- [BS98] Horst Bunke and Kim Shearer. A Graph Distance Metric Based on the Maximal Common Subgraph. *Pattern Recognition Letters*, 19(3-4):255–259, 1998.
- [CG97] Scott D. Cohen and Leonidas J. Guibas. The Earth Mover’s Distance: Lower Bounds and Invariance under Translation. Technical Report STAN-CS-TR-97-1597, Stanford University, 1997.

- [GXTL08] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. Image Categorization: Graph Edit Distance + Edge Direction Histogram. *Pattern Recognition*, 41(10):3179–3191, 2008.
- [HHEW04] Lei He, Chia Y. Han, Brian Everding, and William G. Wee. Graph Matching for Object Recognition and Recovery. *Pattern Recognition*, 37:1557–1560, 2004.
- [LO07] Haibin Ling and Kazunori Okada. An Efficient Earth Mover’s Distance Algorithm for Robust Histogram Comparison. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 29(5):840–853, 2007.
- [LWH03] Bin Luo, Richard C. Wilson, and Edwin R. Hancock. Spectral Embedding of Graphs. *Pattern Recognition*, 36(10):2213–2230, 2003.
- [PM99] Apostolos N. Papadopoulos and Yannis Manolopoulos. Structure-Based Similarity Search with Graph Histograms. In *Proceedings of the International Workshop on Database and Expert Systems Applications (DEXA)*, pages 174–178, 1999.
- [RTG98] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. A Metric for Distributions with Applications to Image Databases. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 59–66, 1998.
- [SKK04] Thomas B. Sebastian, Philip N. Klein, and Benjamin B. Kimia. Recognition of Shapes by Editing Shock Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 26(5):550–571, 2004.
- [SWG02] Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. Algorithmics and Applications of Tree and Graph Searching. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 39–52, 2002.
- [Tak04] Yoshimasa Takahashi. Chemical Data Mining Based on Non-terminal Vertex Graph. pages 4583–4587, 2004.
- [Ume88] Shinji Umeyama. An Eigendecomposition Approach to Weighted Graph Matching Problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5):695–703, 1988.
- [VL05] Fabien Viger and Matthieu Latapy. Efficient and Simple Generation of Random Simple Connected Graphs with Prescribed Degree Sequence. In *Proceedings of the International Computing and Combinatorics Conference (COCOON)*, pages 440–449, 2005.
- [ZWS96] Kaizhong Zhang, Jason T.-L. Wang, and Dennis Shasha. On the Editing Distance Between Undirected Acyclic Graphs. *International Journal of Foundations of Computer Science*, 7(1):43–58, 1996.

Lightweight Performance Forecasts for Buffer Algorithms

Sebastian Bächle and Karsten Schmidt
Databases and Information Systems Group
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
{baechle,kschmidt}@cs.uni-kl.de

Abstract: Buffer memory allocation is one of the most important, but also one of the most difficult tasks of database system administration. Typically, database management systems use several buffers simultaneously for various reasons, e.g., disk speed, page size, access behavior. As a result, available main memory is partitioned among all buffers within the system to suit the expected workload, which is a highly complex optimization problem. Even worse, a carefully adjusted configuration can become inefficient very quickly on workload shifts. Self-tuning techniques automatically address this allocation problem using periodic adjustments of buffer sizes. The tuning itself is usually achieved by changing memory (re-)allocations based on hit/miss ratios, thereby aiming at minimization of I/O costs. All techniques proposed so far observe or simulate the buffer behavior to make forecasts whether or not increased buffer sizes are beneficial. However, database buffers do not scale uniformly (i.e., in a linear fashion) and simple extrapolations of the current performance figures can easily lead to wrong assumptions. In this work, we explore the use of lightweight extensions for known buffer algorithms to improve the forecast quality by identifying the effects of varying buffer sizes using simulation. Furthermore, a simple cost model is presented to optimize dynamic memory assignments based on these forecast results.

1 Introduction

Dynamic database management gained a lot of attention and visibility during recent years and led to various self-tuning approaches. As I/O reduction is one of the most important aspects, automatized buffer memory management has always been one of the building blocks for (self-)tuning of database systems. Data placement decisions but also variations in access patterns, page sizes, access speed, read/write characteristics, or prices of storage devices suggest the support of multiple buffers to optimally exploit existing I/O bandwidth. Memory partitioning, however, frequently entails memory waste, because some buffers may be underused while others are overused. Here, only continuous monitoring of system performance may assure adequate usage of the total memory budget and regular adjustment of buffer allocations at runtime, thereby enabling minimization of waste.

The decision when and which buffers have to be resized requires a cost-based model together with buffer techniques (i.e., page mapping, propagation algorithm) that are self-tunable at runtime. The quality of a decision depends on the cost model itself and the

accuracy of forecasts. However, database buffers typically scale non-uniformly (i.e., in a non-linear fashion) and simple extrapolations of current performance figures can easily lead to wrong assumptions. In the worst case, the redistribution of buffer memory results in unintended buffer sweeps followed by excessive I/O thrashing, which again increases the time to pour oil on troubled waters. In our opinion, self-tuning components should therefore follow a strict “Don’t be evil” policy.

Most tuning approaches aim at maximum speedup, i.e., they focus on the identification of the greatest profiteer when more buffer memory can be assigned. Accordingly, they usually shift memory from buffers having low I/O traffic and/or low potential for performance gains to more promising ones. We believe that a sole focus on buffer growth is dangerous, because the risk of wrong decisions comes mainly from the inaccuracy of forecasts concerning smaller buffers. Once a buffer is shrunk too much, it may cause a lot of I/O and, in this way, also affect the throughput of all remaining buffers. Thus, reliable estimations for buffer downsizing are obviously as important as estimations for buffer upsizing. Good forecast quality is further urgently needed in dynamic environments which have to cope with many or intense workload shifts. Here, too cautious, i.e., too tiny adjustments, even when they are incrementally done, are not good enough to keep the system in a well performing state. Reliable forecasts help to justify more drastic reconfigurations which may be necessary to keep up with workload shifts.

1.1 Forecast of Buffer Behavior

Proposed forecast models for the performance of a resized buffer can be divided into two groups: The first group uses heuristics-based or statistical indicators to forecast buffer hit ratios, whereas the second group is based on simulation. Using heuristics-based approaches, the forecast quality is hard to determine. As a consequence, their use comes with the risk of wrong tuning decisions which may heavily impact system performance. Simulation-based approaches allow trustworthy estimations, but usually only limited to the simulated buffer size. Outside already known or simulated ranges, hit ratios may change abruptly. For this reason, we need forecasts for *growing and shrinking* buffers.

The performance of a buffer does not scale linearly with its pool size, because mixed workloads containing scans and random I/O can cause abrupt jumps in the hit-ratio trend line as illustrated in Figure 1. These jumps may also lead to differing speed-ups for varying buffer sizes, which again may cause wrong assumptions and decisions.

Performance prediction is always based on information gathered by monitoring, taking samples or (user) hints into account. Hit/miss ratios are the standard quality metrics for buffers, because they are cheap and express the actual goal of buffer use: I/O reduction. Unfortunately, they are useless for performance forecasts, i.e., they even do not allow to make simple extrapolations for growing or shrinking buffer sizes. To illustrate this fact, let us assume the following scenario for a given buffer size of 5 and LRU-based replacement. At the end of a monitoring period, we observed 5 hits and 10 misses. At least two different access patterns may have led to these statistics:

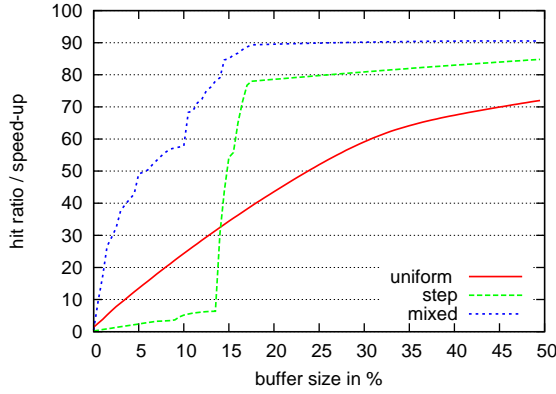


Figure 1: Buffer speed-up trend for different access patterns.

Scenario 1: 1, 2, 3, 4, 5, 1, 1, 1, 1, 1, 6, 7, 8, 9, 10, ...

Scenario 2: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, ...

In the first scenario, 5 hits are attributed to repeated accesses of page 1, whereas, in the second scenario, the hits are attributed to 5 different pages (1, 2, 3, 4, 5). For the same scenarios and a buffer of size 2, we get completely different hit (h) and miss (m) statistics:

Scenario 1: $m, m, m, m, m, m, h, h, h, h, m, m, m, m, m, \dots$

Scenario 2: $m, m, m, m, m, m, m, m, m, m, m, m, m, m, m, \dots$

Obviously scenario 1 obtains a better hit rate with 4 hits to 11 misses than scenario 2 without any hit. If we increase the buffer instead to hold 6 pages in total, the picture turns again:

Scenario 1: $m, m, m, m, m, h, h, h, h, h, m, m, m, m, m, \dots$

Scenario 2: $m, m, m, m, m, h, h, h, h, h, h, h, h, h, h, \dots$

Now we observe 5 hits to 10 misses for scenario 1 and 10 hits to 5 misses for scenario 2. This example shows that hit/miss numbers or page/benefit metrics do not allow for correct extrapolations, because the order of page requests and the hit frequency distribution are important. Thus, self-tuning relies on monitoring and sampling of data where current buffer use is taken as an indicator for the future. Information relevant for resizing forecasts such as re-use frequencies, working set size, or noise generated by scans cannot be expressed in single numbers.

Instead, the ideal starting point for buffer forecasts is the replacement algorithm used for a buffer. Its statistics incorporate a lot more information about these relevant aspects than

any other performance marker. Today, substantial research has already been performed to develop adaptive replacement algorithms, hence, it is safe to assume that such algorithms are “optimally” operating for the available memory. The question is now how to leverage this implicit knowledge for performance forecasts. As we will demonstrate in the remainder of this paper, it is difficult but not impossible to get accurate estimates for buffer downsizing. In combination with already known simulation methods for the estimation of buffer upsizing, we can then build a lightweight framework for dynamic buffer management.

1.2 Related Work

Optimal buffer management has been a key aspect in database system research since the very early days. Thus, various aspects such as the underlying disk model, search strategies within a buffer, replacement algorithms, concurrency issues and the implications of the page layout have been intensely studied [EH84]. Nevertheless, the complexity of buffer management did not allow to distill an optimal configuration for all different kinds of workloads and system environments. Instead, self-tuning mechanisms were explored to resolve performance bottlenecks at runtime.

One early self-tuning approach hints at specific access patterns like scans or index traversals to the buffer to optimize victim selection [JCL90]. This allows to outperform standard LRU-based algorithms but addresses only a single aspect of dynamic buffer management. In [NFS95], the authors give a theoretical base for the combined analysis of buffer sizing decisions and the influence of access patterns. [Dia05] models buffer load balancing as a constrained optimization problem and investigates the application of control theory and optimization theory.

In [SGAL⁺06], control theory, runtime simulation, and cost-benefit analysis are integrated into a self-tuning framework. The presented forecast technique SBPX serves also as our baseline and is introduced in detail in Section 2. Some heuristic forecast techniques are presented in [BCL93, MLZ⁺00]. The analytical work in [THTT08] derives an equation to relate miss probability to buffer allocation. Finally, [DTB09] proposes a brute-force step-by-step approach to determine the optimal configuration for an entire DBMS.

1.3 Contribution

In this work, we study two major prerequisites for self-tuning buffer memory allocation: cost determination and decision making. As the main objective of buffer tuning is I/O reduction and main memory management, decisions based on I/O costs are required to efficiently distribute available memory among all buffer pools. In particular, we look at overhead and quality for buffer undersizing and oversizing forecasts to estimate I/O costs for alternative configurations.

We present ideas to integrate low-overhead forecast capabilities for several common buffer algorithms and assess their feasibility in experiments. Furthermore, we show how these forecasts can be used for nearly riskless self-tuning decisions. Eventually, a short evaluation is revealing prospects of simulation-based buffer tuning as well as its limitations.

The remainder of this paper is organized as follows: Sections 2 and 3 discuss forecast techniques for buffer upsizing and downsizing, respectively. In Section 4, we present a decision model for a self-tuning component. The results of our experiments are shown in Section 5. Finally, Section 6 concludes the paper.

2 Forecast of Buffer Upsizing

The obvious way of accounting I/O costs for alternative buffer sizes is to fully simulate each of them for the same page reference string, i.e., page request sequence. Of course, a simulation of the propagation behavior for page numbers is sufficient; the actual payload data need not be kept in memory. Nevertheless, this approach requires additional data structures, such as hash maps for lookup, lists for the replacement algorithm, and virtual pages. Moreover, each buffer request has to be processed multiple times, i.e., page lookup and replacement maintenance for each simulated configuration. Obviously, the overhead of such a solution is prohibitive. In contrast, cheaper solutions may be less accurate, but still achieve meaningful results for resizing decisions.

Our buffer self-tuning refinements are inspired by the SBPX framework [SGAL⁺06], which approximates the benefit of a larger buffer through “buffer extension”. This extension is simply an overflow buffer for the page identifiers of the most recently evicted pages. The overflow buffer must, of course, have its own strategy for victimization. The authors of SBPX recommend here a strategy “similar to that of the actual buffer pool” [SGAL⁺06].

When a page miss in the actual buffer occurs, the extension checks if the page identifier is found in the overflow buffer, i.e., if the page would have been present in a larger buffer. In that case, we can account a “savings” potential for upsizing. Further, we must now maintain the overflow buffer. The page identifier of the actual evicted page is promoted to the overflow buffer, which in general requires to evict another page identifier from the overflow buffer. This replacement is not exactly the same as a real miss in the simulated larger buffer. The identifier of the requested page causing the miss could have been present in the larger buffer. In the course of continuous requests, however, also a larger buffer must evict pages. Thus, a replacement in the overflow buffer can be regarded as a “delayed” replacement effect. In the case of a page hit in the actual buffer, no further bookkeeping is required, because the locality principle suggests that the replacement strategy in a larger buffer holds a superset of the pages present in a smaller one. Listing 1 shows a sketch of the modified page fix routine.

The problem of this approach is that replacement decisions for two separate buffers in combination are not necessarily the same as for a single large buffer. Thus, the forecast quality of upsizing simulations depends on one aspect: When a page is evicted from the

actual buffer and promoted to the overflow area, we must be able to transfer “state” information (e.g., hit counters, chain position, etc.) from the actual replacement strategy into the overflow strategy (lines 17 and 20). Otherwise, the overflow strategy behaves differently.

Listing 1: Modified page fix algorithm for upsize simulation

```

1 Frame fix(long pageNo) {
2   Frame f = mapping.lookup(pageNo);
3   if (f != null) {
4     strategy.refer(f);           // update replacement strategy
5     ...                          // and statistics
6   } else {
7     Frame of = overflowMapping.lookup(pageNo);
8     if (of != null) {
9       overflowMapping.remove(of.pageNo);
10      ...                          // update overflow hit statistics
11    } else {
12      of = overflowStrategy.victim();
13      overflowBuffer.remove(of.pageNo);
14      ...                          // update overflow miss statistics
15    }
16
17    Frame v = strategy.chooseVictim();
18    strategy.copyStateTo(overflowStrategy);
19
20    v.copyStateTo(of);           // transfer page identifier to overflow
21    overflowMapping.put(of.pageNo, of);
22
23    mapping.remove(v.pageNo);
24    ...                          // replace page in frame v
25    strategy.referAsNew(v);      // update replacement strategy
26    ...                          // and statistics
27    mapping.put(pageNo, v);
28  }
29 }

```

3 Forecast of Buffer Downsizing

As shown above, knowledge about the performance gain through a larger buffer is useful to determine the greatest profiteer of more memory among several buffers. However, the question for the buffer(s), which may be safely shrunk without suffering from severe penalties, remains unanswered. The authors of SBPX extrapolated downsizing costs as the inverse of savings potential gained through upsizing [SGAL⁺06]. For buffer sizes close to the unknown (!) borders of working set sizes, however, this bears the risk of wrong decisions. Therefore, we developed a simple mechanism to find out if page hits would have been also page hits in a smaller buffer. In combination, the SBPX technique allows us now to determine which buffer profits the most from additional memory, while our approach helps us to determine which buffer suffers least from downsizing.

The goal of buffer replacement algorithms is the optimized utilization of data access locality, i.e., to keep the set of the currently hottest pages that fits into memory. Accordingly, a small buffer is assumed to keep an “even hotter subset” of the pages that would be present in the actual buffer. Based on this assumption, we denote a subset of the pages in a buffer

of size n as $hotset_k$, if it would be kept in a smaller buffer of size k . The key idea of our approach is to keep track of this hotset during normal processing. When a page is found in the buffer and belongs to the hotset, it would have been a hit in the smaller buffer, too. However, if a requested page is in the current buffer but not in the hotset, the smaller buffer would need to evict another page, which must be, of course, part of the current hotset and load the requested page from disk. Here, we only have to maintain the hotset. The page that would have been evicted from the smaller buffer is removed from the hotset and the requested page is added to the hotset. Each swap is accounted as a page miss for the simulated smaller buffer.

Of course, a page miss in the current buffer would also be a page miss in a smaller buffer. Accordingly, we have to select a replacement victim for both the current buffer and the (simulated) smaller buffer. The real victim page is now replaced with the new page and swapped with the virtual victim of the smaller buffer into the hotset. The modified page fix algorithm is shown in Listing 2.

Listing 2: Modified page fix algorithm for downsize simulation

```

1 Frame fix(long pageNo) {
2   Frame f = mapping.lookup(pageNo);
3   if (f != null) {
4     if (!f.hotSet) {
5       Frame v = strategy.chooseHotSetVictim();
6       f.hotSet = true;           // swap frame to hotset
7       v.hotSet = false;
8       strategy.swapHotSet(f, v);
9       ...                       // update simulated statistics
10    }
11    strategy.refer(f);           // update replacement strategy
12    ...                          // and statistics
13  } else {
14    Frame v = strategy.chooseVictim();
15    mapping.remove(v.pageNo);
16    ...                          // replace page in frame v
17    if (!v.hotSet) {
18      Frame hv = strategy.chooseHotSetVictim();
19      hv.hotSet = false;         // swap frame to hotset
20      v.hotSet = true;
21      strategy.swapHotSet(f, v);
22    }
23    strategy.referAsNew(v);       // update replacement strategy
24    ...                          // and statistics
25    mapping.put(pageNo, v);
26  }
27 }

```

Note that a real replacement victim is generally not expected to be part of the current hotset, because this would imply that the replacement strategy evicts a page more recently accessed. In some algorithms, however, such counter-intuitive decisions might be desired, e.g., to explicitly rule out buffer sweeps through large scans. Then, we must not maintain the hotset at all.

Obviously, the overhead of this approach is very small. We only need a single bit per buffer frame to flag the hotset membership and must determine a swap partner, when a new page enters the hotset. Furthermore, the simulation does not influence the quality of the current buffer, i.e., the strength of the replacement strategy is fully preserved. As said, the choice of the hotset victim is dependent on the used replacement strategy to reflect the behavior of

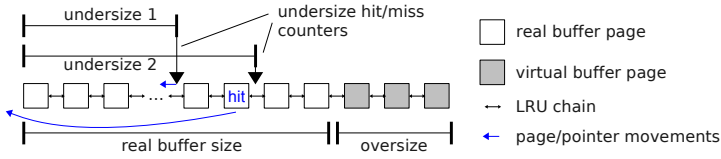


Figure 2: LRU-based buffer simulation with overflow extension

the strategy in a smaller buffer correctly. In the following, we will investigate hotset victim determination for four popular families of replacement algorithms. In particular, we want to know if it is possible to predict replacement decisions for a smaller buffer based on the implicit knowledge present.

3.1 LRU

The LRU algorithm embodies a very simple, yet effective replacement strategy. It evicts always the *least recently used* page from a buffer. Typically, it is implemented as a doubly-linked list as shown in Figure 2.

On request, a page is simply put to the head of the chain. Thus, LRU finds its replacement candidate always at the tail. Accordingly, the last k pages of the LRU chain in a larger buffer of size n are identical with the k pages in the simulated smaller buffer of size k and the hotset victim page is found at the k -th position from the head. The overhead of pointer dereferencing to position k can be avoided with marker pointer, which is cheap to maintain. Hence, the hotset victim is guaranteed to be identical to the victim as in the smaller buffer and the simulation is fully precise. Evidently, the simplicity of LRU even allows to easily simulate at the same time the effects when the current buffer would be reduced to different smaller sizes, which is especially useful for precise step-wise tuning decisions. It is sufficient to place a marker at each desired position.

3.2 LRU-K

The LRU-K algorithm [OOW99] follows a more general idea of LRU and takes the last K references of a page into account. By doing so, it is “scan-resistant” and less vulnerable to workloads where frequently re-used pages mix with those having hardly any rereference. For each page, LRU-K maintains a history vector with the last K references and the timestamp of its last reference. Furthermore, history vectors of already evicted pages are retained for re-use if an evicted page is requested again within the so-called *retained information period* (*RIP*). The replacement victim is only searched among those pages that have been buffered for at least a predefined *correlated reference period* (*CIP*). The rationale behind this idea is to prevent a drop of pages immediately after their first reference. For further details on *CIP*, history maintenance, etc., we refer to the original paper.

The victim page is determined by the maximum backward K -distance, i.e., the page with the earliest reference in the history vector. Thus, although implemented differently, LRU-K behaves for $K = 1$ the same as LRU. The hotset victim is chosen accordingly as shown in Listing 3. Note that implementations of LRU-K usually maintain a search tree for that. For simplicity, we present here the modification of the unoptimized variant as in the original paper.

Due to the history update algorithm described in [OOW99], more than one victim candidate can exist. This could become a problem for our simulation, because a real buffer might choose a different victim than simulated. Therefore, we simply evict the candidate with the least recent reference (line 12). As the timestamp of the last access is unique, our simulation will be accurate here. Instead, the choice of *RIP* turns out to become a problem. If the garbage collection for history entries is not aligned, pages that re-enter the smaller buffer will be initialized differently than in simulation, which may affect future replacement decisions.

Listing 3: LRU-K hotset victim selection

```

1 Frame hotSetVictim() {
2   long min = t;
3   long minLast = Long.MAX_VALUE;
4   Frame v = null;
5   for (int i = 0; i < pages.length; i++) {
6     Frame p = pages[i];
7     History h = p.history;
8     if ((p.hotSet) && (t - last > CIP)) {
9       long last = h.last;
10      long dist = h.vector[k - 1];
11      if ((dist < min)
12          || ((dist == min) && (last < minLast))) {
13        victim = p;
14        min = hist.vector[k - 1];
15      }
16    }
17  }
18  return v;
19 }

```

3.3 GCLOCK

The third strategy is GCLOCK [NDD92], which stands for *generalized clock* algorithm. Like LRU-K, it takes the reference history of a page into account. In contrast to LRU-K, however, it is likely to degrade through scans but can be implemented with less computational and space overhead. The buffer itself is modeled as a circle of buffer frames, i.e., the clock. Each frame also maintains a simple reference counter, which is incremented for each reference to that specific page. For victim selection, the “clock hand” circles over all frames and decrements the reference counters. The clock hand stops at the first frame where the reference counter drops below zero. So, frequently referenced pages remain longer in the buffer, because they have higher reference counts.

The determination of a hotset victim is straightforward: We simply have to circle over the frames and look for the first hotset page whose reference counter would first drop below zero. Obviously, this is the page with the minimum reference counter. The algorithm is sketched in Listing 4.

Listing 4: GCLOCK hotset victim selection

```

1 Frame chooseHotSetVictim()
2 {
3     Frame v = null;
4     int h = clockHand;
5     for (int i = 0; i < size; i++) {
6         Page p = circle[(++h % size)];
7         if (p.hotSet) {
8             if (p.count == 0) {
9                 return v;
10            } else if ((v == null) || (p.count < v.count)) {
11                v = p;
12            }
13        }
14    }
15    return v;
16 }

```

Again, this only approximates the behavior of a smaller buffer with GCLOCK. There are two reasons: First, the angular velocity of the clock hand in a smaller buffer is higher because there are less frames. Second, the circular arrangement of buffer frames makes the algorithm inherently dependent on the initial order. Thus, victim selection is not only a matter of the page utilization, but also a matter of clock-hand position and neighborhood of frames. Using a second clock hand (i.e., pointer) walking solely over the hotset frames is necessary to address differing round trips. However, swapping of frame positions when the hotset is maintained would impact behavior of GCLOCK in the actual buffer – a circumstance, we want to avoid. To improve forecast quality, we implemented the smaller circle, i.e., the hotset, with forward pointers for hotset pages that point to the logical next one. In case of swapping (see lines 8 and 21 in Listing 2), only the forward pointer and a hotset counter for that page need to be maintained. In Section 5, we will show that these minor efforts can lead to almost perfect estimations.

3.4 2Q

The 2Q algorithm [JS94] is a simplified way of imitating LRU-2, which is noted for delivering good hit ratios but often poor performance due to its complex algorithm. In essence, 2Q is a combination of FIFO and LRU. On the first reference, 2Q places a page in a FIFO queue (denoted *a1*). The first re-reference of a page in the *a1* queue promotes it to the LRU chain (denoted *am*). The effect of these two “stages” is that only hot pages are promoted to the LRU chain, which tends to keep cold pages longer than necessary. These cold pages, i.e., pages that are accessed only once within a longer time period are now dropped earlier by the FIFO queue. An extended version of 2Q splits the FIFO queue to keep track of rereferences to pages evicted from the FIFO queue [JS94]. The effect is similar to the history caching of LRU-K and comes with queue sizing problems for forecasts, too.

Sizing problems also arise for the FIFO queue and the LRU chain in the standard algorithm. Therefore, we used a simplified variation of 2Q where all buffer frames are assigned to the LRU chain and the FIFO queue only stores references to the pages in the LRU chain. So, it serves like an index for the LRU chain to identify pages referenced only once so far. Victims are primarily selected from the FIFO queue to replace those pages earlier. A subtlety of 2Q is here that the FIFO queue must not be drained to give new pages a chance for rereference and promotion to the LRU chain. The minimum fill degree of the FIFO queue is a configurable threshold. For simulation, we must therefore count the number of hotset entries in the queue, to be able to decide when a smaller buffer would pick a victim from the FIFO queue and not from the LRU chain. Also, the threshold must be the same for both sizes. Although this results in uniform retention times within the FIFO queue for differing LRU chain sizes, it is acceptable to some degree, because the threshold models the granted window for references of new pages. The hotset victim selection is sketched in Listing 5.

Listing 5: 2Q hotset victim selection

```

1 Frame chooseHotSetVictim()
2 {
3     Frame v;
4     if ((al.numberOfHotsetEntries() > threshold)) {
5         v = al.head();
6         while (!v.hotSet) v = v.alNext;    // Follow FIFO queue to first hotset page
7     } else {
8         v = am.head();
9         while (!v.hotSet) v = v.amNext;    // Follow LRU chain to first hotset page
10    }
11    return v;

```

4 Buffer Tuning

The crucial point in database tuning is the difficulty to precisely predict how a tuning decision will affect system performance. Even experienced database administrators with a deep knowledge of the workload and the database product itself regularly face this challenge. They rely on the assistance of sophisticated monitoring tools to prevent negative effects of their tuning decisions on the production system. Often they also run several observe-analyze-adjust cycles with reference workloads beforehand on dedicated test systems. Of course, this is time-consuming and expensive. Built-in self-monitoring and tuning components can ease this dilemma and reduce the risk of wrong decisions through rather small but continuous and incremental adjustments. In dynamic environments, however, those mechanisms may react too slow to keep up with the rate of workload shifts or short-term resource allocation for higher-level tuning decisions like auto-indexing. Therefore, we aim towards a re-formulation of the central question of automatic tuning from “Which adjustment *certainly* will give the greatest performance benefit?” to “Which adjustment *most likely* will give a performance benefit *but certainly not* result in a performance penalty?”. In other words, when we know that our reconfigurations will not harm, we get the freedom to try quicker and more aggressive tuning options.

In general, the total amount of buffer memory is limited and so the decision to assign more memory to a certain buffer is directly coupled with the decision of taking this memory from one or several others. Fortunately, the performance optimization heuristics for I/O-saving buffers (e.g. data pages, sorting) is straightforward: The more main memory can be used the better. Even an oversized buffer, i.e., a buffer larger than the actual data to be buffered, is less likely to become a performance bottleneck due to bookkeeping overhead. It is just a waste of main memory. The downsizing of a buffer, however, comes along with severe risks: the buffer's locality may drastically decrease and even turn into thrashing causing excessive I/O, which also influences throughput of other buffers. Accordingly, we concentrate on the forecast of the negative effects of memory reallocations and base our tuning decisions not only, as common, on the estimated *benefits*, but also on vindicable forecasts of additional *costs*.

4.1 Cost Model

Automatic tuning needs to derive costs from system state or from system behavior to quantify the quality of the current configuration. Additionally, it also needs to estimate the costs of alternative configurations to allow for comparison. Ideally, these costs comprise all performance-relevant aspects including complex dependencies between system components and future workload demands in a single number to allow for perfect decisions. Clearly, such a perfect cost model does not exist in practice. Instead, costs are typically derived from a mixture of cheaply accounted runtime indicators and heuristics-based or experience-based weight factors. The hope is to reflect at least the correct relationship between alternative setups w.r.t. to performance. The more precise this much weaker requirement can be met, the easier we can identify hazardous tuning decisions before they boomerang on the system.

In contrast to computational costs of a specific algorithm, costs expressing the quality of a buffer are inherently dependent on the current workload. Buffering 5% of the underlying data, for example, can be an optimal use of main memory at one moment, but become completely useless a few moments later. Therefore, each cost value is a snapshot over a window at a certain point in time with limited expressiveness for at most few periods ahead in the future. We define the general goal function for our tuning component: At a given point in time t with a configuration c , find a configuration c' that has less accumulated I/O costs over the next n periods. The optimal window size and the number of forecast periods again depend on the actual workload; slowly changing workloads enable more precise cost estimations for longer periods, while rapidly changing workloads also decrease accuracy of future costs.

For simplicity, our cost model only considers buffer service time, i.e., the time needed to handle a page fix request. Of course, costs assigned to a specific buffer are dominantly determined by the number of I/Os performed. On a buffer miss (denoted m), a victim page has to be selected for replacement and flushed, if necessary, before the requested page is fetched from disk. Accordingly, a buffer miss causes at least one read operation, but may also cause several writes for flushing write-ahead log and victim page. The ratio between

reads and synchronous writes is reflected by a weight factor f_{dirty} , which may vary over time and from buffer to buffer.

Depending on the characteristics of the underlying devices or blocking times under concurrent access, I/O times can also vary between various buffers. Hence, the costs of all buffers must be normalized to a common base to become comparable. We use here a second weight factor w_{buffer} for each buffer. As the time needed for a single I/O operation is easy to measure, these factors can be derived and adjusted at runtime causing low overhead. Finally, the cost of a buffer at the end of time period t is expressed as:

$$c_{buffer}(t) = w_{buffer}(t) \cdot (1 + f_{dirty}(t)) \cdot m(t)$$

Note, we assume that CPU costs can be safely ignored, either because they are independent of whether an operation can be performed on buffered data or requires additional I/O, or because additional CPU cycles for search routines in larger buffers are negligible compared to an I/O operation. In the remainder of this paper, we also assume that read and write operations have symmetric costs and a low variance. However, it should be evident that the presented basic model can be easily extended to take asymmetric read/write costs (e.g. for solid state drives), different costs for random and sequential I/O, and also the apportionment of preparatory, asynchronous flushes of dirty pages into account.

4.2 Decision Model

Our buffer balancing is based on the cost model of Section 4.1. In certain intervals, the buffer configuration is analyzed and optimized if main memory reallocations are promising reduced I/O costs for the entire system.

After each monitoring period, the buffer pools are ranked by their cost estimations as follows. The higher a buffer pool is ranked in the *save* list, the more costs can be saved (i.e., this equals to providing a higher benefit) by referring to the simulated buffer oversize. On the other hand, buffer pools are also ranked by their cost estimations for undersize figures, whereas the minimum cost increase is ranked top in the *rise* list. Using a greedy algorithm, buffer pool pairs are picked from the top of both lists as long as the cost reduction on the *save* list is higher than the increase on the *rise* list. Note, a buffer may occur in both lists, which typically indicates a “jump” and is thereby easily recognized. Finally, resize mechanisms are employed to perform the memory “shifts”. The selected buffer from the *save* list is increased to allow more frames and references to be cached. A buffer chosen from the *rise* list, however, is shrunk, which may also include to flush victims to achieve a smaller buffer size. Note, an optimal solution is always achievable, but certainly requires more efforts. Therefore, we use greedy optimization because it is fast, cheap, and fairly good.

Oversize and undersize simulations for several buffer pools do not necessarily have the same size in bytes, which complicates memory shifts. However, fine-grained assignments may be required, which are also possible by extrapolating the buffer scaling figures between its real size and the simulated sizes.

To avoid thrashing, buffers chosen for resizing are removed from both ranking lists. The simulated undersize and oversize areas have to be adjusted as well, which is similar to the “regular” buffer resize. For instance, the number of hotset pages is reduced by selecting victims out of this subset and by switching their flags. Obviously, oversize areas can be kept or resized as desired.

Although resize decisions are sometimes heavy-weight operations (e.g., flushing pages), they only occur at the end of each monitoring period and are only performed as long as expected benefits justify them.

Period Refinements for Simulated Buffer Sizes

Accounting hit/miss numbers for multiple simulated and real buffer sizes over a certain period of time induces estimations errors. For instance, a smaller buffer causing more misses requires more time to process the same amount of buffer requests as the real one. On the other hand, a larger buffer having an improved hit ratio may require less time to process the requests, which are considered during this simulation and tuning period. Therefore, simulation-based cost accounting has to reuse the cost model’s I/O weights for read and write operations to adjust the (simulation) periods. That means, undersize simulation has to limit I/O accounting as soon as the I/O budget that is physically possible is consumed and vice-versa for SBPX extensions.

Switchable Propagation Algorithms

Adjusting memory assignments for buffer pools is also limited to the scalability prospects of a specific buffer algorithm. However, different buffer algorithms may perform differently and exchange of an algorithm would be an alternative tuning option without actually shifting memory. But different algorithms tend to use manifold figures such as access counters, timestamps, or history queues. The major problem is to carry over the current information when switching to a new algorithm. A poor alternative is to reset the entire propagation strategy. However, a practical way is to initialize the new algorithm by evicting all the “old” pages into the new algorithm and continue to use the new algorithm. The decision to switch the algorithm can only be based on a full simulation of an alternative propagation algorithm relying on a similar cost model as presented in Section 4.1.

5 Evaluation

Here, we want to evaluate the accuracy of our extensions as well as the decision quality for buffer balancing. But first, we have to describe our benchmark scenarios and their workloads.

As already stated in the introduction, buffers do not scale uniformly; thus, we generated reference strings for various (common) scenarios including random and sequential access of varying sizes.

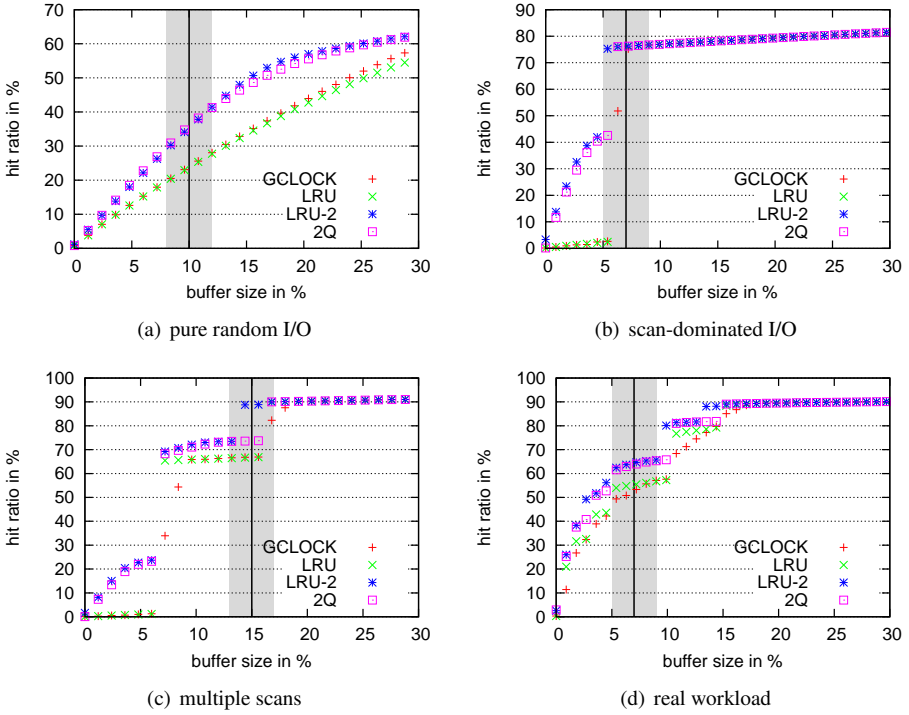


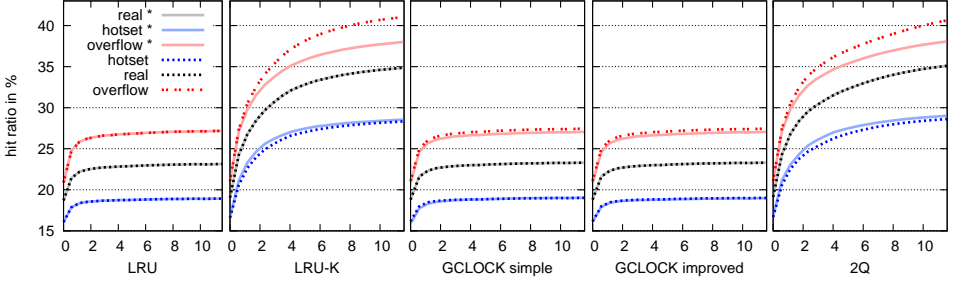
Figure 3: Buffer scalability for various workloads and replacement algorithms

5.1 Workload

In Figure 3(a)-3(d), we analyze the critical buffer size ranges for various access patterns whose characteristics are summarized in Table 1. Note, the total number of DB pages is equal to the first column’s object size figure of each scenario in this table. The only uniformly scaling buffer is measured for workloads dominated by random I/O (see Figure 3(a)), where the overall hit ratio is – as expected – quite low. In this case, re-sizing extrapolations will work properly, but such an access behavior is unusual in databases. Dominating scans mixed with random access are modeled and measured in Figure 3(b). Although scan resistance is addressed by replacement algorithms, scan effects easily provoke “jumps” in the buffer performance. In such cases, the buffer hit rate dramatically increases as soon as often occurring scans entirely fit into the buffer. Such “jumps” remain undetected if monitoring happens only at one side of the “jump”. The third workload shown in Figure 3(c) is a mixture of multiple scans and random accesses in a single buffer. This scenario may represent a more typical buffer usage pattern which exhibits a realistic buffer scaling. In Figure 3(c), several areas can be identified having different slopes, where each area boundary may cause uncertainty for extrapolations. In the last sample workload shown in Figure 3(d), a mixture of high-locality scans and some share of random accesses is analyzed. This typical workload scenario causes several (small) “jumps” resulting in a

Table 1: Workload characteristics

workload	Figure 3(a) (random)		Figure 3(b) (scan)		Figure 3(c) (jumps)			
request share in %	50	50	25	75	10	65	25	
\sum object size (pages)	150k	22k	150k	7k	150k	7k	13k	
access type	rnd	rnd	rnd	seq	rnd	seq	seq	
workload	Figure 3(d) (real)							
request share in %	10	10	10	20	10	20	10	10
\sum object size (pages)	250k	5k	10k	10k	500	500	1k	2k
access type	rnd	rnd	seq	seq	seq	seq	seq	seq

Figure 4: Estimation accuracy for workload *random* (buffer calls $\times 100.000$ on x-axis)

stair-case pattern. In this case, fine-grained extrapolations necessary for buffer tuning may quickly fail, although the slope in the average is quite similar.

We want to show in the subsequent sections that our algorithms are capable of identifying and handling all of these (more or less) typical workload scenarios.

5.2 Accuracy

The quality of buffer balancing is based on the estimation quality of our extended buffer algorithms. Therefore, we need to evaluate the estimation accuracy for the differing workloads. For the following experiments, the gray-shaded areas in Figures 3(a)–3(d) specify the simulated ranges centered around the actual buffer sizes indicated by the black lines. For simplicity, we always use a fixed range of $\pm 2\%$ of the total DB size. For each workload, we measure the undersize and oversize estimation accuracy. Each of the Figures 4–7 contains the results of five algorithms using the same workload and up to 1.2 Mio buffer calls. The lines marked with an asterisk (*) illustrate the simulation-based hit ratios and, to enable comparison, the others show those of real buffers having the same sizes.

The first graphs are always showing the standard LRU behavior, which is always delivering perfect estimation accuracy; however, its hit ratio performance is not the best. But its lightweight simulation is definitely a plus. In contrast, the LRU-K results (second graphs) constantly indicate top hit ratios but show weaknesses in forecast quality. Especially, the downsize simulation of the *scan* workload fails with a dramatic overestimation.

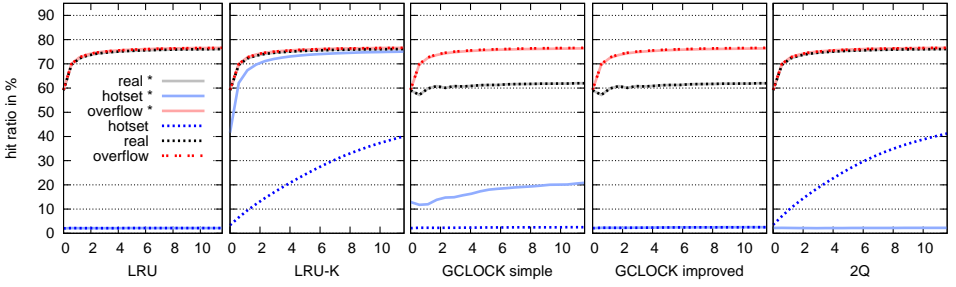


Figure 5: Estimation accuracy for workload *scan* (buffer calls $\times 100.000$ on x-axis)

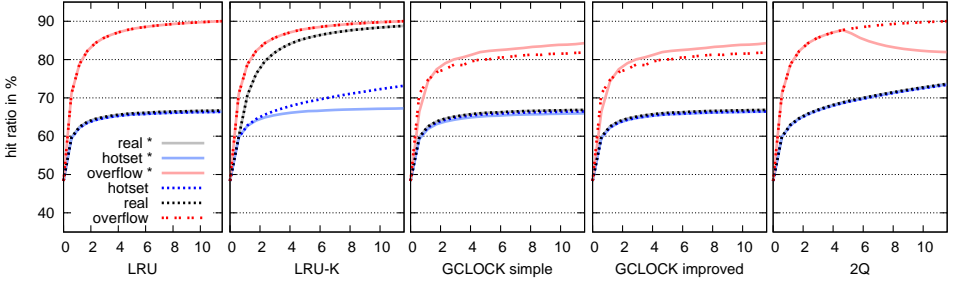


Figure 6: Estimation accuracy for workload *jumps* (buffer calls $\times 100.000$ on x-axis)

The results for GCLOCK in Figure 5 and 7 (third graphs) reveal its sensitivity to page order and clock-hand position for hotset simulations. By adding a second clock hand and forward pointers to simulate a separate clock for the hotset pages, we achieve considerably better accuracy (fourth graph), but its performance is always behind all other strategies.

On the right-hand side, we measure the forecast quality provided by the simplified 2Q algorithm. In all scenarios, it delivers top results while only requiring low maintenance overhead. However, forecast quality is disappointing in some scenarios. Similar to LRU-K, it fails for workload *scan*, but in the opposite direction with underestimation. Further, we observe a suddenly degrading forecast quality for the workloads *jumps* and *real*. Even worse, oversize estimations as well as undersize estimations are affected. Even the use of a separate policy for the oversize buffer does not lead to better results.

The experiments reveal that our simulations based on the locality principle lead to trustworthy estimations in many cases. On one side, simple algorithms like LRU and GCLOCK fit well into our framework. On the other side, more advanced algorithms such as LRU-K and 2Q also allow lightweight estimations, but suffer from unpredictable estimation errors in some scenarios. The reasons are built-in mechanisms to achieve scan-resistancy, which are hard to model in simulations. Further, these algorithms do not allow logical composition of individual buffers.

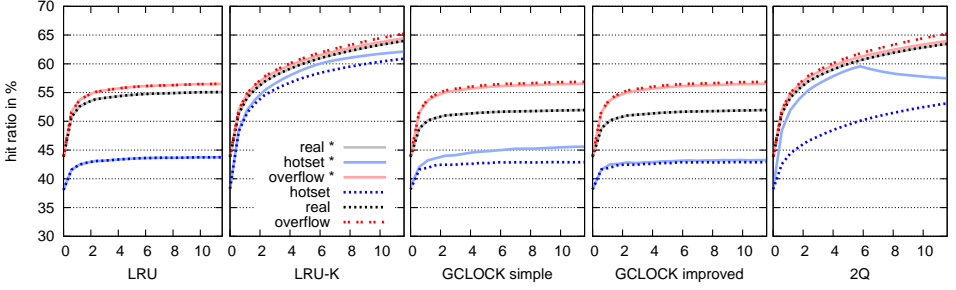


Figure 7: Estimation accuracy for workload *real* (buffer calls $\times 100.000$ on x-axis)

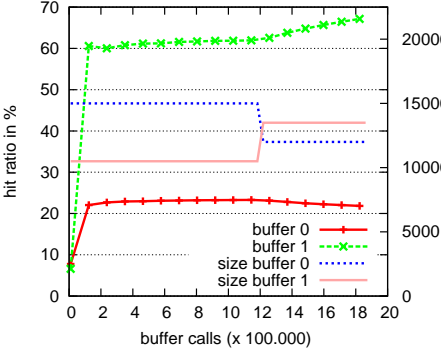


Figure 8: Buffer balancing *random* vs. *scan*

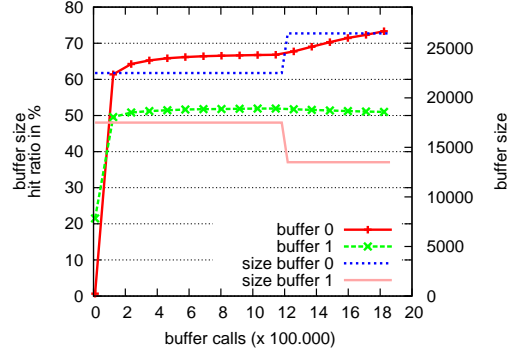


Figure 9: Buffer balancing *jump* vs. *real*

5.3 Buffer Balance

In Figure 8, the self-tuning mechanism presented in Section 4.2 automatically tunes two buffers, where buffer 0 was fed with *random* workload from Figure 3(a) and buffer 1 with *scans* shown in Figure 3(b). Buffer sizes (i.e., simulation and real) are chosen as described in Section 5.2. Due to space limitation, we present the results only for the improved GCLOCK and a fixed memory shift granularity of 2 % of the DB size. After the buffers were warmed up (i.e., after 1.2 Mio buffer calls), the cost model triggers all memory shifts. The *random* workload buffer was shrunk according to its hotset simulation, whereas buffer 1 was increased. Although the hit ratio of buffer 0 slightly descends, the overall I/O performance improves, because the hit ratio of buffer 1 increases considerably.

Because the self-tuning decisions are based on a cost model, they are applicable for arbitrary scenarios. In our second example, we again use two buffers, one that is fed from the *jumps* workload generator and the other from the *real* workload generator as shown in Figure 3(c) and Figure 3(d). In this setting, SBPX fails because it does not recognize that the size of buffer 0 is close to a “jump” boundary. However, as indicated by Figure 9, our downsizing simulation detects the pitfall and prevents buffer performance penalties.

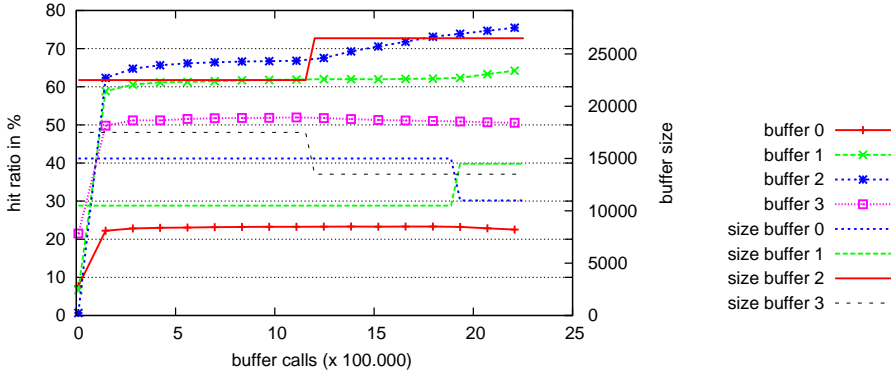


Figure 10: Balancing of four buffers under different workloads

Resizing two buffers is obviously simple. Therefore, we combine both experiments in a single setup shown in Figure 10. The cut-out shows two memory shifts leading to minor descends of the hit ratio on the one side but clear improvements on the other side resulting in a steadily improved buffer performance.

In summary, we could experimentally prove that buffer balancing can be achieved at low cost, but it heavily depends on accurate and lightweight forecasts for both directions – upsize and downsize.

6 Conclusions

Even after decades of research on buffer management and optimization, the problem of a reliable, dynamic adaptation of buffer memory allocation is not fully solved. In this work, we studied opportunities to forecast buffer resizing effects to support harm-free self-tuning decisions. As downsizing a buffer is accompanied with severe risks of thrashing, we argued that reliable prediction of downsizing effects is a key point for self-tuning decisions. Furthermore, we argued that additional overhead for these forecasts must not add noticeable overhead to normal processing. Therefore, we focused on lightweight techniques to exploit knowledge from the buffer replacement strategies for forecasts and presented possible solutions for four families of replacement algorithms.

In our experiments, we could show that forecast quality is heavily dependent on the actual strategy. It seems that sophisticated strategies like LRU-K and 2Q make it hard or even impossible to get reliable forecasts for either upsizing, downsizing, or both. We found that there are two reasons for this: First, such algorithms use history-recording techniques, which are very costly to emulate for varying sizes. Second, they are extremely sensible to configuration parameters, which cannot be easily negotiated between differing buffer sizes. However, simpler, yet widely-used strategies like LRU and GCLOCK turned out to allow for cheap and highly accurate or even perfect forecasts. In conjunction with a

simple cost model and a greedy algorithm, we demonstrated the use of forecasts to improve buffer hit ratios without the risk of severe performance penalties. Following the idea of differing “stages” in 2Q to improve buffer behavior, our findings suggest to think about further partitioning of buffers with complex replacement strategies into several distinct buffers with simpler but more predictable strategies. This way, forecasts would generally become reliable and fragmentations issues were automatically resolved by the self-tuning capabilities.

References

- [BCL93] K. P. Brown, M. J. Carey, and M. Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *VLDB 1993*, pages 328–341. Morgan Kaufmann, 1993.
- [Dia05] Y. Diao *et al.* Comparative Studies of Load Balancing With Control and Optimization Techniques. In *ACC '05: Proc. 24th American Control Conf.*, pages 1484–1490, 2005.
- [DTB09] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTunes. *Proc. VLDB Endow.*, 2(1):1246–1257, 2009.
- [EH84] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.
- [JCL90] R. Jauhari, M. J. Carey, and M. Livny. Priority-Hints: An Algorithm for Priority-Based Buffer Management. In *VLDB 1990*, pages 708–721, 1990.
- [JS94] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB 1994*, pages 439–450, 1994.
- [MLZ⁺00] P. Martin, H.-Y. Li, M. Zheng, K. Romanufa, and W. Powley. Dynamic Reconfiguration Algorithm: Dynamically Tuning Multiple Buffer Pools. In *DEXA 2000*, pages 92–101, 2000.
- [NDD92] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing. In *ACM SIGMETRICS 1992*, pages 35–46, 1992.
- [NFS95] R. Ng, C. Faloutsos, and T. Sellis. Flexible and Adaptable Buffer Management Techniques for Database Management Systems. *IEEE Trans. Comput.*, 44(4):546–560, 1995.
- [OOW99] E. J. O’Neil, P. E O’Neil, and G. Weikum. An Optimality Proof of the LRU- Page Replacement Algorithm. *Journal of the ACM*, 46(1):92–112, 1999.
- [SGAL⁺06] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *VLDB 2006*, pages 1081–1092, 2006.
- [THTT08] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. H. Tung. A new approach to dynamic self-tuning of database buffers. *Trans. Storage*, 4(1):1–25, 2008.

Offline Design Tuning for Hierarchies of Forecast Models

Ulrike Fischer, Matthias Boehm, Wolfgang Lehner

TU Dresden; Database Technology Group

Abstract: Forecasting of time series data is crucial for decision-making processes in many domains as it allows the prediction of future behavior. In this context, a model is fit to the observed data points of the time series by estimating the model parameters. The computed parameters are then utilized to forecast future points in time. Existing approaches integrate forecasting into traditional relational query processing, where a forecast query requests the creation of a forecast model. Models of continued interest should be deployed only once and used many times afterwards. This however leads to additional maintenance costs as models need to be kept up-to-date. Costs can be reduced by choosing a well-defined subset of models and answering queries using derivation schemes. In contrast to materialized view selection, model selection opens a whole new problem area as results are approximate. A derivation schema might increase or decrease the accuracy of a forecast query. Thus, a two-dimensional optimization problem of minimizing the model cost and model usage error is introduced in this paper. Our solution consists of a greedy enumeration approach that empirically evaluates different configurations of forecast models. In our experimental evaluation, with data sets from different domains, we show the superiority of our approach over traditional approaches from forecasting literature.

1 Introduction

In many domains, gathered data constitutes *time series*, e.g., sales per month, system load per hour, energy supply per minute. This is especially valid in data warehouse systems, where the time dimension is virtually guaranteed to be present [KR02]. This data is often used as a basis of decision-making processes. Forecasting is a fundamental prerequisite for such decisions, otherwise all decisions rely on the history only and might not be valid. One important use case is forecasting of energy supply. Many renewable energy sources (e.g., solar panels) pose the challenge that production depends on external factors (e.g., amount of sunlight). Hence, available power can only be predicted but not planned, which makes it rather difficult for energy distributors to efficiently include renewable energy sources into their daily schedules. This problem addresses the MIRACLE project [BBD⁺10] that predicts the energy demand and supply of customers and suppliers and balances accordingly. This poses the challenge of high query and update intervals (15-minutes or less), where forecast queries need to be answered as fast and accurate as possible.

There are existing approaches that integrate time series forecasting into traditional relational query processing in DBMS [DB07, Ora10, Pre10]. In contrast to exporting the data to an external statistical program, these approaches allow for joint query processing and

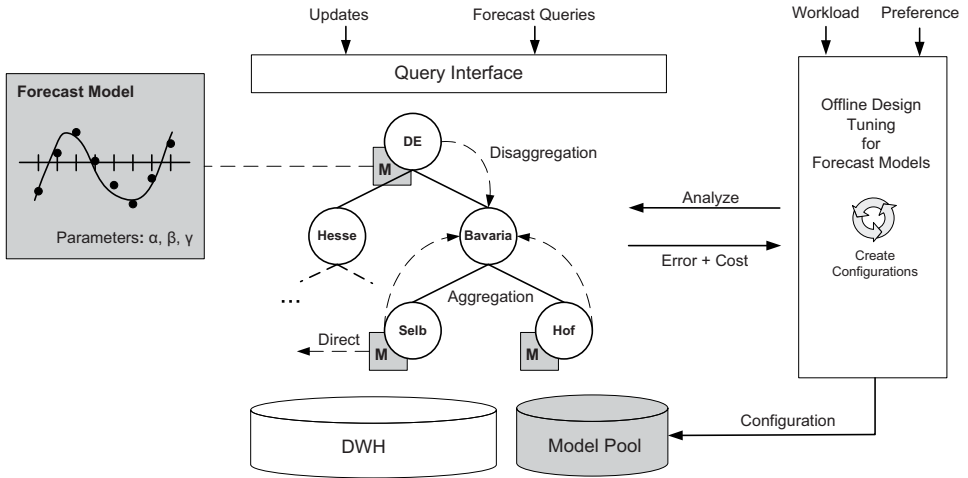


Figure 1: System Overview

exploit database specific optimization techniques. In this context, a *forecast query* is specified like a traditional query extended with a forecast horizon, which specifies the number of values to forecast or a future point in time [DB07, FRBL10]. A forecast query uses a model of the time series at hand to calculate the expected future behavior of the time series. In general, model-based forecasting involves two phases, *model creation* and *model usage*. Model creation tries to fit a model (e.g., represented by the parameters of a continuous function) to the observed data points of the original time series (Figure 1 left). Model creation is typically computationally expensive, often involving numerical optimization schemes to estimate the d model parameters that span a d -dimensional search space. In contrast, model usage utilizes the parameters calculated in the first step to forecast future points of the observed time series. It is cheap as only a few simple operations are necessary. Due to the high model creation costs, query processing can be sped up if models are only built once and kept in a *model pool*. Subsequent queries are answered by choosing a fittable model from this pool [FRBL10, ACL⁺10, GZ08].

However, as a data-warehouse might contain a high number of individual time series, building a model for each single time series is expensive. In addition, time series characteristics change over time, requiring maintenance in form of parameter re-estimation. Only a few forecast methods however allow updating the parameters analytically by using just the new time series values, most approaches require access to the complete historical time series for parameter re-estimation. Parameter re-estimation might be as expensive as model creation. Therefore, in domains like energy supply, where minute-by-minute data is stored and forecasted, we do not have enough time to keep all models up-to-date until the next query arrives. One solution to this problem is to choose a well-defined subset of forecast models. Forecast queries can then be answered by specific derivation schemes.

There are two main derivation schemes in the area of forecasting – aggregation and disaggregation [Fli01]. *Aggregation* calculates the forecast values of a time series by using

forecast values of subset time series, while *disaggregation* uses the forecast values of another time series representing a superset, e.g., by using the historical fraction. In the center of Figure 1 a simplified hierarchy for forecasting energy supply in Germany (abbr. DE) is presented. Here, the energy supply of single cities is recorded at level one. The supply is then aggregated according to different regions (level two) and to the supply over the whole country (level three). Now, the forecast values for the region Bavaria could be either calculated by disaggregation from the forecast values of the total time series over Germany or by aggregation over the forecast values of the single cities Selb and Hof.

However, the accuracy of a forecast value calculated from a model specifically created for a given query might be different from the accuracy of a forecast value derived from models at different aggregation levels of the time series. Interestingly, some derivation schemes might even improve the accuracy. Therefore, in addition to model usage and maintenance cost, we need to minimize the forecast error of queries using this model, resulting in a two-dimensional optimization problem. However, the error of a model cannot be determined without actually building the concerning model [DWD76, ACL⁺10]. As a result, any solution to this problem requires the empirical comparison of alternative configurations.

In general, this problem seems similar to materialized view selection and usage. However, model selection requires a second metric, the forecast accuracy, while materialized view selection focuses on minimizing query and maintenance cost only. For model usage, forecast queries always output approximated tuples, so we can exploit additional derivation schemes, i.e., disaggregation. On the other hand, forecast models are always created at instance level of the data, so we can not apply compensation queries, e.g., it is impossible to calculate a selection on top of a model while this is a valid usage of materialized views.

To summarize, our offline design tuning algorithm takes as input a workload and a user preference regarding execution time and accuracy. It creates different configurations of models and empirically analyzes their forecast error and maintenance cost (Figure 1 right). As a result, the best configuration for the given workload is provided to the model pool. This configuration leads to a reduction of query processing times as models are already present in the database, a possible higher forecast accuracy as aggregation dependencies are taken into account and less maintenance costs as only necessary models are stored.

Contributions and Outline In summary, we make the following contributions:

- First, we introduce the fundamentals of physical design of forecast models in a data-warehouse schema with multiple defined hierarchies (Section 2.1 and 2.2).
- We then define our two-dimensional optimization problem, i.e., increase forecast accuracy and reduce maintenance cost (Section 2.3).
- Third, we present our greedy enumeration approach to reduce the space of possible configurations (Section 3.1) and present heuristics, which might reduce the number of forecast models considered (Section 3.2).
- Finally, we show the applicability of our approach in an experimental evaluation (Section 4).

We will finish with related work in Section 5 and conclude in Section 6.

2 Multi-Hierarchical Forecasting

In this section, we first sketch the basics of a multi-hierarchical forecasting system. Subsequently, we explain the notion of physical design in such a context. We finish by discussing conflicting optimization challenges and by formulating our general optimization goal.

2.1 Multi-Hierarchical Forecasting System

Hierarchical forecasting is based on grouping time series into groups, group families and so on [Fli01]. Each level results from the aggregation of the child elements one level below. The top level is the total aggregate of all elementary time series. If we transfer the hierarchical forecasting approach to a data-warehouse environment, where multiple hierarchies exists in parallel, merged by foreign keys to dimensions in the fact table, we get a *multi-hierarchical forecasting system*. This generalization also contains different levels of aggregation, where a parent element is calculated by aggregation of corresponding child elements on an arbitrary level below. However, as we have multiple hierarchies, a child might contribute to several parents. In addition, it might be possible to calculate parents' values from multiple sets of child nodes on the same level.

Definition 1 Multi-Hierarchical System: *In a multi-hierarchical system, the value $S_{ij}(t)$ at time t of the time series S_{ij} with index j at level i is calculated as follows:*

$$S_{ij}(t) = AGG_{l \in G_{pij}} S_{(i-1)l}(t),$$

where G_{pij} contains a list of child indexes of group p at level $i - 1$, which contribute to the aggregate of element S_{ij} . AGG is an aggregation function, e.g., SUM . Elements at level 1 are the elementary time series, while the element at level h is the total aggregate over all time series. The total number of elements $|S|$ is calculated by the Cartesian product over the number of elements per hierarchy.

Example 1 An example multi-hierarchical forecasting systems is shown in Figure 2. Recall our running example of forecasting energy supply. The energy supply can also be distinguished according to different energy sources, e.g. solar energy. Therefore, in addition to the location hierarchy (Figure 2 left), the energy supply can be aggregated according to different products (Figure 2 right). In the center of Figure 2, these two hierarchies are combined into a multi-hierarchical structure, where a single element represents an instance from both hierarchies (e.g., the supply of wind energy in Hof). Child elements are additionally annotated with an index p (at the top), where elements at level i with the same index can be used to calculate the corresponding parent. For example, the element $S_{31} = R_1$ can be either calculated by aggregating elements $S_{21} = C_1$ and $S_{24} = C_2$ or by aggregating elements $S_{22} = R_1P_1$ and $S_{23} = R_1P_2$.

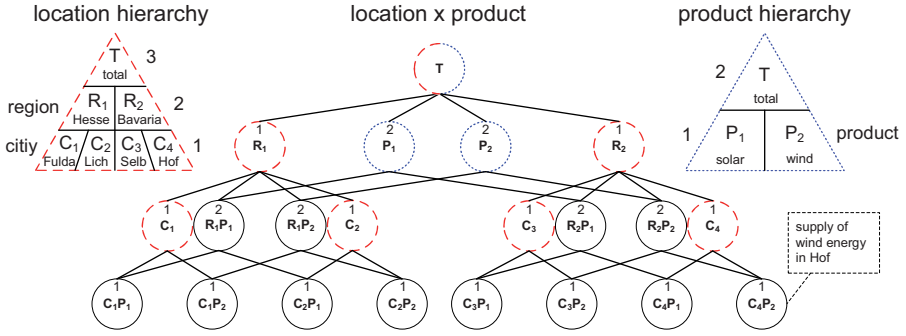


Figure 2: Multi-Hierarchical Forecasting Structure

Note that the multi-hierarchical structure is based on the instance-level of the data. In contrast to the aggregation lattice on attribute level, we include additional functional dependencies that might be given by information assurances (ORACLE) or derived from the underlying data. For example, in Figure 2 instances of the grouping (R, C, P) are not considered, as R directly depends on C and therefore (R, C, P) is equal to (C, P) .

Forecasting Having this structure in mind, for each element S_{ij} , we can calculate the forecast value of the corresponding time series by three different ways:

Forecast Model: First, we can create a forecast model M_{ij} directly from the time series S_{ij} . We can then use this model to directly calculate a forecast value for element S_{ij} .

Aggregation: Second, we can create forecast models for all child elements with the same group index p at level k , where $k < i$. We then forecast using each model in the group and aggregate the forecast values to get the forecast value of the parent element. We denote this strategy as $A_{ij(kp)}$. In addition, we can aggregate recursively.

Disaggregation: Third, we can create a forecast model for one parent element p at level k and disaggregate the forecast value. The disaggregation strategy requires the calculation of a disaggregation key $D_{ij(kp)}$. A simple, but quite successful disaggregation strategy (assuming SUM as aggregation method), is an average over the fraction of the child series and the parent series: $D_{ij(kp)} = 1/n \cdot \sum_{t=1}^n S_{ij}(t)/S_{kp}(t)$ [GS90]. Then, the forecast values of the series element S_{ij} are the product of the disaggregation key $D_{ij(kp)}$ and the forecast value of S_{kp} .

Each forecasting strategy allows different underlying methods. For example, we could use exponential smoothing as forecast method, the summation as aggregation method and an average over the fraction of child and parent series as disaggregation method. While choosing a concrete method is independent from our approach, it might have a high impact on the resulting physical design.

2.2 Physical Design

Conceptually, the user expects a forecast model for every time series queried. However, based on the three possibilities to calculate the forecast values for a single element S_{ij} , we can have different physical designs in a multi-hierarchical system. The benefit of a physical design depends on the workload of the system. For example, a model at a higher level might support long-term forecasts as the general trend of the data is captured. We therefore consider a workload trace of forecast queries for a given period of time. A workload W consists of elements S_{ij} , their relative frequency f and the corresponding forecast horizon h : $W = \{S_{ij}, f, h\}$. An element S_{ij} might be either described by point queries or pure conjunctive queries, i.e., addressing multiple hierarchies. However, queries might also address several elements S_{ij} (e.g., disjunctive, join or group-by queries). In that case, we only store the individual elements in our workload model.

Definition 2 Configuration: *For a given workload W , a configuration C_W is a valid assignment of forecast models to individual elements. An assignment is valid if each element S_{ij} in W can be calculated by either a forecast model, aggregation or disaggregation.*

In order to make sure that we can calculate forecast values for each element S_{ij} in W , we always include the *least common parent* in our configuration, i.e., the element (might not be part of W) at the smallest level that is parent of all (other) elements in W .

Example 2 Recall Example 1 and consider the workload $W = \{(R_1, 1/5, 1), (C_1, 1/5, 1), (C_2, 1/5, 1), (C_1P_1, 1/5, 1), (C_1P_2, 1/5, 1)\}$, where the one-step ahead forecast for the supply in the region Hesse (R_1), the supply in the city Fulda (C_1), the supply in the city Lich (C_2), the supply of solar energy in Fulda (C_1P_1) and of wind energy in Fulda (C_1P_2) is requested. A first valid configuration is to create a model over the supply in the region Hesse $\{R_1\}$ and calculate all workload elements by disaggregation. A second possible configuration is $\{C_1, C_2\}$ that calculates elements C_1P_1 and C_1P_2 by disaggregation and element R_1 by aggregation. Many more possibilities exist.

It is obvious that the number of possible configurations is exponential with the number of possible models $|C_W|$. For each element, we can decide if we create an individual forecast model and we can choose an arbitrary combination of models.

2.3 Optimization Problem

Our goal is to find the best configuration for a given workload. Each configuration can be described by two metrics, configuration error and configuration maintenance cost.

Intuitively one would think that a forecast value directly calculated from a model is always superior than disaggregation from a higher level model. However, many studies in mathematical and forecasting literature have shown that disaggregation (or aggregation) can be superior to individual time series models [DWD76, Fli99]. Therefore, the forecast error is

not monotonic (related to a hierarchical system), as adding a new forecast model does not automatically imply a lower forecast error.

Definition 3 Configuration Error: *Given a configuration C_W , the error E_W is calculated by the total error over all accessed elements in W using the best strategy:*

$$E_W = \sum_{(S_{ij}, f, h) \in W} f \cdot \min_{\forall k_1 > i, k_2 < i} (e(M_{ij}, h), e(D_{ij(k_1)}, h), e(A_{ij(k_2)}, h)) \quad (1)$$

where M_{ij} denotes a forecast model, $D_{ij(k_1)}$ denotes the disaggregation and $A_{ij(k_2)}$ denotes the aggregation strategy. We use the time series of length $|S_{ij}| - h$ to train the model M_{ij} and calculate the disaggregation key D_{ij} . Then, $e(\text{strategy}, h)$ calculates the error for the given strategy over the h -step-ahead forecast over the remaining time series.

The forecast error is calculated using the absolute value of the symmetric mean absolute percentage error (SMAPE), which is a scale-independent accuracy measure. For both, aggregation and disaggregation, the forecast error is determined by choosing the set of child elements or the parent with the minimal forecast error. Note that we do not support direction changes, i.e., a disaggregate cannot be calculated from an aggregate as this might lead to an arbitrary result, not following the characteristics of the time series.

Maintenance of forecast models requires mainly parameter re-estimation, as maintaining the state of a model (e.g., smoothing constants) is cheap. The costs of parameter re-estimation depends on factors like frequency of re-estimation, time series length and runtime of the parameter estimation algorithm. However, as we are in an offline context, we assume a fixed estimation method and strategy leading to equal costs for each element. In addition, we can assume that the time series are about the same length as aggregate series at higher levels need all their child series to be of the same length. However, if a series has a missing value, this value might even have a meaning (e.g., zero products sold this month). Finally, we do not include the maintenance of disaggregation keys as these are, similar to the state of the model, significantly cheaper than parameter re-estimation of models. Following these considerations, we explicitly use a very simplified model to estimate the maintenance cost of a configuration by using the number of created models.

Definition 4 Maintenance Cost: *Given a configuration C_W , the maintenance cost B_W are calculated by the number of forecast models in C_W :*

$$B_W = |M_{ij} \in C_W|. \quad (2)$$

Now, our optimization goal is twofold: First, we want to reduce the forecast error. Second, we want to do as less maintenance as possible.

Definition 5 Optimization Goal: *Our optimization goal is to find the configuration C_W , which is minimal according to the configuration error E_W and maintenance costs B_W :*

$$\min_{C_W} \left(\alpha \cdot \frac{E_W}{E_T} + (1 - \alpha) \cdot \frac{B_W - 1}{B_{max} - 1} \right) \text{ with } B_{max} > 1 \text{ and } \alpha \in [0, 1], \quad (3)$$

where B_{max} as well as E_T are used as normalization constants. B_{max} is the maximum number of models in C_W and E_T is the forecast error for the configuration, where only a model for the common parent is used. For normalization purposes as well, we subtract 1 in the second part of the equation.

With parameter α we can weight the importance of both dimensions. If we set $\alpha = 0.5$, we give equal weight to maintenance cost and forecast error. If we set $\alpha = 1$ we try to find the best configuration according to the forecast error, without regarding maintenance. This generalizes the problem of finding the best hierarchical structure in forecasting literature. As the forecast error is not monotonic with respect to the number of models, the minimum forecast error can result for any configuration. On the other hand, with $\alpha = 0$, as we do not consider disaggregation costs, we get the configuration where only one model is used for the common parent and all other elements use the disaggregation strategy.

3 Offline Design Approach

In order to solve the optimization problem of Definition 5, our general approach works as follows:

1. Create a forecast model for each element in the workload and the common parent.
2. Enumerate all valid configurations according to Definition 2.
3. Evaluate each configuration regarding the configuration error (Definition 3) and maintenance cost (Definition 4).
4. Choose a configuration according to Definition 5 and drop all models, which are not part of the solution.

However, this naive algorithm has unacceptable runtime, because we first need to create a forecast model for every element considered in the workload. This is necessary to calculate the model, aggregation and disaggregation benefit for each element. However, model creation is expensive as parameters need to be estimated. Second, it is infeasible to enumerate all possibilities as the number of possible configurations is exponential with the number of elements (on instance-level). Therefore, we present an approach to reduce the number of configurations enumerated (Section 3.1). Then, we discuss some heuristics to reduce the number of forecast models considered (Section 3.2).

3.1 Greedy Enumeration

As explained before, maintenance cost increase monotonically with each additional forecast model while the behavior of the forecast error is unknown. A new model might lead to an improvement but the error could also stay similar or get even worse. Thus, in the

Algorithm 1 Greedy Enumeration

Require: *elementsConsidered*

```
1: bestConf  $\leftarrow$  concat(1,repeat(0, numElements - 1))
2: bestEval  $\leftarrow$   $\alpha$ 
3: repeat
4:   stop  $\leftarrow$  true
5:   currentConf  $\leftarrow$  bestConf
6:   for i in elementsConsidered do
7:     currentConf[i]  $\leftarrow$  1
8:     if (currentEval = evaluateConfiguration(currentConf)) < bestEval then
9:       bestConf  $\leftarrow$  currentConf
10:      bestEval  $\leftarrow$  currentEval
11:      newModel  $\leftarrow$  i
12:      stop  $\leftarrow$  false
13:    currentConf[i]  $\leftarrow$  0
14:  elementsConsidered  $\leftarrow$  remove(elementsConsidered, newModel)
15: until stop = true
```

second case, we would never use this model as only maintenance increases. Therefore, we propose a greedy enumeration approach, where we start with the configuration where only a model for the top element is used. Thus, this is the valid configuration with minimal maintenance cost. We then try to add additional models step-by-step, using most promising models first. As each model is described with equal maintenance costs, we always add the model next, which results in the lowest overall configuration error.

Algorithm 1 outlines our greedy enumeration approach. First, our algorithm requires the elements, which actually can qualify for a model. This could contain all elements in the multi-hierarchical structure. However, the workload will reduce these elements significantly to only the queried elements. In addition, the heuristics introduced in the next section will reduce the number of models considered further. To represent a configuration, we use a vector where 1 stands for forecast model and 0 for no model. The entries are the consecutive elements in the multi-hierarchy starting at the top and going down from left to right. Therefore, in line 1 we create the configuration, where only a model is created for the top element. According to Definition 5, the evaluation for our start configuration is always α , which we use as our current best evaluation (line 2). Then, for each element in our list of considered elements, we create a configuration where a model is used for this element (line 7). We evaluate this configuration with Definition 5 and if it is better than the current one, we store it (line 8-10). Then, we reset the current configuration (line 13). Therefore, in each step we iterate over all elements and output a new configuration with one additional element. In the end, we remove the new element from the list of considered elements (line 14). We stop when we find no better configuration than the current one. Note that we need to check the benefit of each single element in each iteration as a new model might have an impact on every other element in the workload.

Example 3 To illustrate the greedy approach, an example is shown in Figure 3. This example is created for the workload described in Example 2 with $\alpha = 0.5$. A gray box shows

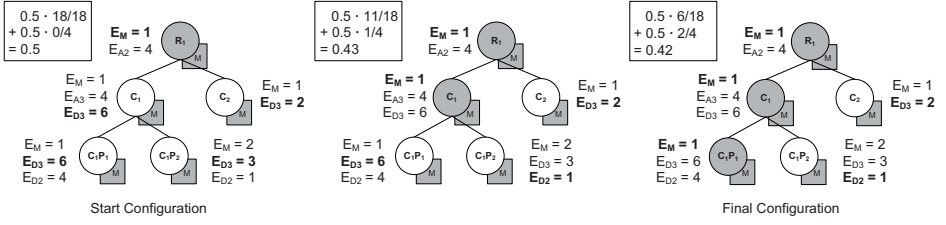


Figure 3: Greedy Enumeration

that an evaluation model was created for the corresponding element, while a gray colored node implies that the model is actually used and maintained in the current best configuration. Each node is annotated with the errors when using a model E_M , aggregation E_{A_k} and disaggregation E_{D_k} , where k is level from which the forecast values are aggregated or disaggregated. In the left part of Figure 3, the start configuration is shown in which only a model for the top element is used. As explained before, the start evaluation equals always α , which is shown in the box in the upper left corner (Definition 5). Note that we use the total error to describe the configuration error as all elements in the workload have the same frequency. Now, our greedy approach sequentially checks the benefit of a model for each element. Elements C_1 and C_1P_1 have the highest disaggregation error of 6.0 and would both profit from a model equally as the model error is only 1.0. However, element C_1P_2 would also profit from a model at element C_1 as the disaggregation error using level two is lower. Therefore, we would decide to use a model for element C_1 (Figure 3 center). The new evaluation drops down to 0.43. Second, the greedy approach would decide to use a model for the element C_1P_1 as we get the best improvement regarding the forecast error and the evaluation drops to 0.42. Although, the element C_2 would also lead to a decrease of the error, the algorithm is finished now. The improved error does not compensate the increased maintenance costs. The evaluation would be 0.51.

For ease of illustration, we used a single hierarchy in this example. However, the only difference in a multi-hierarchy is that a child node might benefit from the disaggregation from several parents and a parent from the aggregation of several sets of child elements.

This approach might result in local sub-optima for two reasons. The first reason is that models are never removed, although they might not be necessary anymore as the element can be calculated by aggregation. Therefore we additionally analyze aggregation benefits. Thus, whenever we evaluate a configuration, we also check if we can remove models, which can now be calculated by aggregation. If so, we also evaluate the configuration with the removed aggregate model. Second, we might miss optimal configurations because we only consider single models with each step. However, a whole group might improve the result as elements at higher levels can be calculated by aggregation. Therefore, we include a second optimization, where we analyze groups in each step as well. In our case, a group is a complete set p of elements at one level, which contribute to the same aggregate one level above. Different strategies to build groups are possible. However, this can have the drawback that groups are added too early in the process resulting in a different local sub-optima. In addition, we still end in local sub-optima if adding more than one model (but

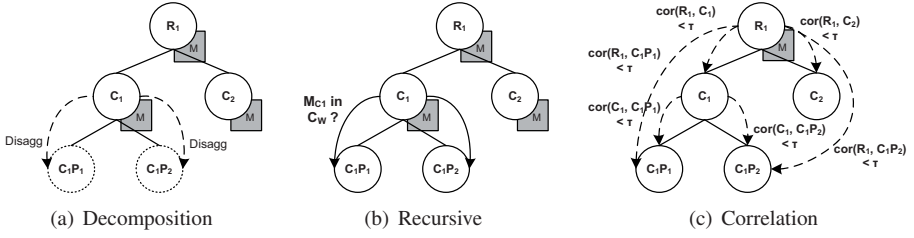


Figure 4: Model Creation Heuristics

not a whole group) would allow the removal of an aggregation model and lead to a better overall forecast error. However, in most cases, aggregation benefits are less important as they require many child models to be build.

The complexity of the greedy enumeration approach is $O(n + n^2)$ in the worst case. First, we create n forecast models. Then, we evaluate for each element the benefit of a model and add the element with the most benefit. We stop when we can not find a beneficial model anymore resulting in a worst case evaluation of $\sum_{i=1}^n i \approx n^2$ configurations. In the best case, we stop after one run as we did not find a better configuration, so we result in a linear behavior. Note that the creation of a forecast model might be much more expensive than the evaluation of one configuration.

3.2 Heuristics

In this section, we present heuristics to reduce the number of forecast models considered. Each heuristic only creates a subset of possible models. If there is no model created for an element, our greedy enumeration approach does never consider a configuration where a model for this element is used (Algorithm 1, line 6). All heuristics can be used by themselves in addition to the greedy enumeration approach, but an arbitrary combination of these heuristics is also possible.

Decomposition The decomposition approach assumes that if we find a good strategy for each single hierarchy, the combination of hierarchies, i.e., conjunctive queries, will also result in a low error. Therefore, only forecast models are considered that address elements from single hierarchies. For all combinations of hierarchies we always use the disaggregation strategy (Figure 4(a)). This heuristic reduces the number of considered elements to the sum over the number of elements in each single hierarchy.

Recursive The recursive approach has the underlying assumption that if disaggregation is best for one level, it is also best for all underlying child elements. Initially, we only create forecast models for the top element and the level underneath. Every time we decide to use a forecast model, we create all models one level below (Figure 4(b)). In the best case, this approach only creates models for the top two levels of the hierarchy. However, in the worst case, all models are created. If we combine the recursive heuristic with the decomposition

heuristic, we might even get a higher reduction of forecast models created, as each child node is only reachable through exactly one parent node.

Time Series Characteristics This heuristic analyzes the characteristics and the relation of the time series to filter out forecast models to be created and considered. For this, we use two different characteristics, correlation and disaggregation error. With both approaches, forecast models are only created if a certain characteristic is fulfilled.

Correlation The core observation is that high correlated time series follow the same pattern and thus could be calculated by the same model. Therefore, we calculate the correlation between parent-child relations. For this, we also consider parent-child relations over more than one level (Figure 4(c)). Then, we only create models for child series if the correlation is below a threshold τ . The threshold τ defines the aggressiveness of this approach. A low τ might filter out many models, but might also miss good configurations. In contrast, a high τ is safer, but might only filter out a few models.

Disaggregation Error As we always create the top model, we can analyze the disaggregation error of each element in the workload. This approach only creates a model for an element if the disaggregation error is above ω . To calculate ω , we use the median of the disaggregation error $\text{median}(e(D_{ij(11)}, h))$ over the elements in the workload and the weight α of the configuration error. Then, ω is calculated by $\text{median}(e(D_{ij(11)}, h))/(\alpha + 0.5)$. Therefore, if we give equal weight to the configuration error and maintenance cost ($\alpha = 0.5$), we create all models higher than the medium disaggregation error. This works as we assume equal maintenance cost in this paper. If we give low weight to the configuration error, we create less models as ω increases and vice versa. Therefore, we adjust the number of created models according to the weight of the configuration error as a lower weight would result in a configuration with less models anyway.

4 Experimental Evaluation

We conducted an experimental study in order to evaluate (1) the performance of our approach on three data sets from different domains with respect to traditional approaches from forecasting literature, (2) the performance and scalability of the proposed heuristics and (3) the adaptability to different user requirements.

4.1 Experimental Setting

To implement the described offline design tuning approach we used the statistical computing software environment R. It provides efficient build-in forecast methods and parameter estimation approaches, which we used to build the individual forecast models. All experiments were executed on an IBM Blade (Suse Linux, 64bit) with two processors (each a Dual Core Intel Xeon at 2.80 GHz) and 4 GB RAM.

In order to show the general applicability of our offline design approach, we use the fol-

	electricity	tourism	energy
#elements level 1	1,568	32	86
#elements level 2	273	12	1
#elements level 3	14	1	-
#elements level 4	1	-	-
series length	28	25	5,808

Table 1: Sizes of Data Sets

lowing three data sets:

- *Electricity: Worldwide Electricity Generation* The first data set was obtained from the US Energy Information Administration and is public available at [US110]. This data set includes metered world-wide electricity generation. It consists of two hierarchies. The first hierarchy contains regional information from world-wide over continental to individual countries. In the second hierarchy different categories of electricity sources such as renewable or nuclear are distinguished. After merging both hierarchies, we result in multi-dimensional hierarchy with four levels, similar to Figure 2. The data is available from 1980 until 2008 in an annual resolution.
- *Tourism: Australian domestic tourism* The second data set consists of quarterly observations on the number of visitor nights for the Australian domestic tourism, which is an indicator of tourism activity. The sample begins with the first quarter of 2004 and ends with the first quarter of 2010. The series are obtained from the National Visitor Survey, which is managed by Tourism Research Australia [TRA10]. The data consists of two hierarchies, purpose of visit and state, resulting in three levels of the final multi-hierarchical structure.
- *Energy: EnBW MEREGIO Energy Demand* The third data set was provided by a partner from the MIRACLE project [BBD⁺10] and was obtained during the MEREGIO project [MER10]. This data set contains energy demand from 86 customers ranging from November 1st 2009 to June 30th 2010 in a 1 h resolution. It therefore consists of a single hierarchy with two levels, i.e., level 1 contains the individual customer demand while level 2 contains the total aggregate over all customers.

Table 1 shows the sizes of the different levels and time series lengths for each data set.

For all three data sets we fixed the forecast, aggregation and disaggregation method. As forecast method we use triple exponential smoothing. The class of exponential smoothing methods is widely used in hierarchical forecasting and has proven to be very robust and applicable in an automated fashion to a large set of time series [Cha00]. If we would use a more complex forecast method, we would save even more execution time with our approach as parameter estimation gets more expensive. For the aggregation method, we use summation. Gross and Sohl analyzed 21 disaggregation methods [GS90] and concluded that a simple average of the elements' proportion of the parent element over the entire historical period worked best compared to other, partly more complex, methods. Therefore, we will use this as disaggregation strategy.

In addition, we fix the workload. We assume that each element in the hierarchy is queried once requesting a one-step ahead forecast, so $W = \{(S_{ij}, 1/|S|, 1)\}$ for $i=1 \dots \#levels$, $j=1 \dots \#elements \text{ of level } i$ and $|S|$ is the total number of elements in the whole multi-hierarchical structure. Therefore, we analyze the worst case, where every element is considered equally. A different workload would only reduce the search space.

4.2 Performance Comparison

In the following, we compare our greedy approach with the three traditional approaches "bottom-up", "top-down" [Fli01] and "complete". The bottom-up approach creates forecast models for all time series at level one and calculates all other forecasts by aggregation of elementary forecasts. The top-down approach creates only one forecast model for the top element and calculates all other forecasts by disaggregation of the top forecast. Finally, the complete approach creates forecast models for all elements in the hierarchy and calculates all forecasts using directly the model for the concerning element. In this experiment, we use 80% of the data sets to learn the models for all approaches and to learn the configuration for our greedy approach. We set $\alpha = 0.3$, so we give more weight to maintenance time. Then, we use the remaining 20% to produce one-step ahead forecasts, where we reestimate the model parameters after each forecast. Note that we also could use a more sophisticated model maintenance method, where we trigger parameter reestimation time- or threshold-based. However, this is not the scope of this paper and the effect would be similar for all approaches. Figure 5 summarizes the results for all three data sets and approaches. In Figure 5(a), the average forecast error (using the accuracy measure SMAPE) of each element over the evaluation period is illustrated. Here, the maximum forecast error equals 1. In Figure 5(b), the relative maintenance cost compared to the complete approach as well as the total number of used models is shown.

Our goal is to create as less models as possible while reaching a low forecast error. If we take a rough view at the results, we see that our greedy algorithm produces the best result considering both dimensions. Let us analyze each data set a bit more closely. For the electricity consumption, the complete (C) and bottom-up (B) approach have a much better forecast error than the top-down (T) approach. Therefore, our greedy approach (G) creates 61 forecast models reducing the forecast error a lot but adding only little additional maintenance time. In order to reduce the error even more and beat the complete approach, we need to choose a higher value of α . We will analyze the effect of α more closely in Subsection 4.4. For the tourism activity, the forecast error of the complete and bottom-up approach is slightly better than the top-down approach. However, the forecast error can actually be reduced a lot, if a few forecast models are created at level two and the forecasts at level one are created by disaggregation from level two. Therefore, our greedy approach decides to create some forecast models at level two, reducing the forecast error even more than the complete approach but resulting in much less number of models to maintain. For the energy demand, the top-down approach is similar to the complete and bottom-up approach. Therefore, our greedy approach decides to use the top-down approach, creating no additional forecast models where we save a lot of maintenance cost compared

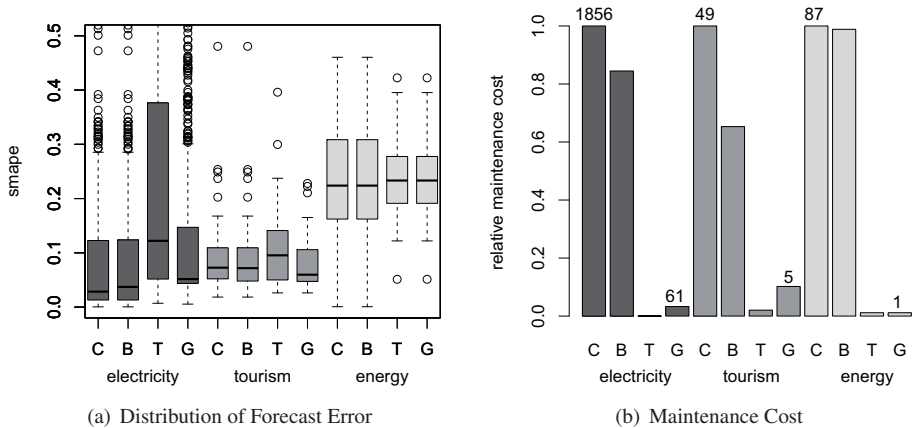


Figure 5: Comparison of Different Approaches

to the complete or bottom-up approach (one instead of 87 models).

To summarize, on the one hand our greedy approach can find a better configuration than the traditional approaches, even with a low value of α . On the other hand, maintenance time is reduced by using only necessary models. The speed up we can achieve strongly depends on the used hierarchy, i.e., the number of elements, and on the relationship of the time series, i.e., weather bottom-up or top-down is superior in general. In terms of total execution time, the benefit strongly depends on the used forecast method and the time series length. For example, for the energy data set, we save about half a minute in each maintenance step if we use triple exponential smoothing. The total maintenance time is 3.4h for the complete approach and 2.3 minutes for the greedy approach. If we use an AR(12) model, which is an instance of the widely used class of ARIMA models, we save 26 minutes in each *single* maintenance step. As we are in an offline context, we did not explicitly measure the speed up of forecast queries. However, a low number of forecast models implies lower maintenance cost and thus, a lower system load or lower query processing times if deferred maintenance is used.

4.3 Comparison of Different Heuristics

In a second experiment, we take a closer look at the different heuristics introduced in Subsection 3.2. For each heuristic, we might get a slightly worse configuration compared to the full greedy approach but we might save execution time. Figure 6(a) shows the *evaluation* (= value of objective function according to Definition 5) of all heuristics compared to the evaluation of the greedy approach. As all data sets exhibit totally different execution times, Figure 6(b) shows only the relative execution time decrease to the full greedy approach. In this experiment, we set $\alpha = 0.75$ so that the impact of the heuristics are higher

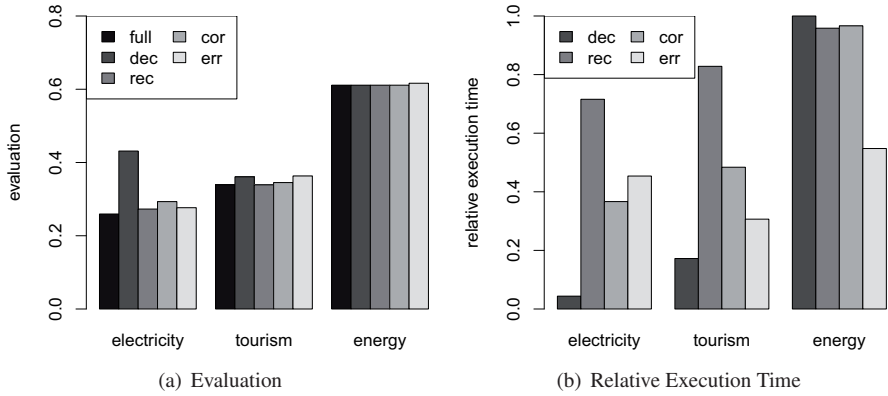


Figure 6: Comparison of Different Heuristics

as more forecast models are created.

The energy data set contains only a single hierarchy. Therefore, the decomposition heuristic (dec) leads to the same accuracy and execution time as the full greedy approach (full). For the other two data sets, it shows the lowest execution time but also increases the evaluation most for electricity and tourism data set. Therefore, it should only be used if very little time is available. The recursive heuristics (rec) increases the evaluation only slightly for the electricity data set but not at all for the other two data sets. However, it shows the smallest improvement in terms of execution time. Therefore, it can be safely used to save some of the execution time of the offline design approach. For the correlation heuristic (cor), we set the correlation threshold to 0.75. This heuristic takes the third rank in terms of execution time and only increases the evaluation slightly for the electricity and tourism data set. As a result, it should be preferred to the recursive heuristic in order to save more execution time. The disaggregation error (err) heuristics shows the second best execution time and only increases the evaluation slightly for all three data sets. To summarize, besides the decomposition heuristic, all heuristics only increase the evaluation slightly compared to the full greedy approach. Most execution time can be saved when the heuristics are used, which take time series characteristics into account.

In order to examine the scalability of our greedy approach and the different heuristics, we vary the size of the electricity data set. For this, we increase the number of electricity sources from one to seven and combine the resulting electricity source hierarchy with the region hierarchy. If we use only one electricity source, we result in a single hierarchical structure, which consists of 464 elements. With each additional electricity source, we add 232 elements. The experimental results are displayed in Figure 7(a). The full greedy approach (using no heuristic) has the longest runtime and shows a super linear behavior with increasing number of elements. In general, all heuristics also increase the number of considered models with increasing data size resulting in a super linear behavior as well. However, the concrete outcome strongly depends on the data itself, e.g., how many time

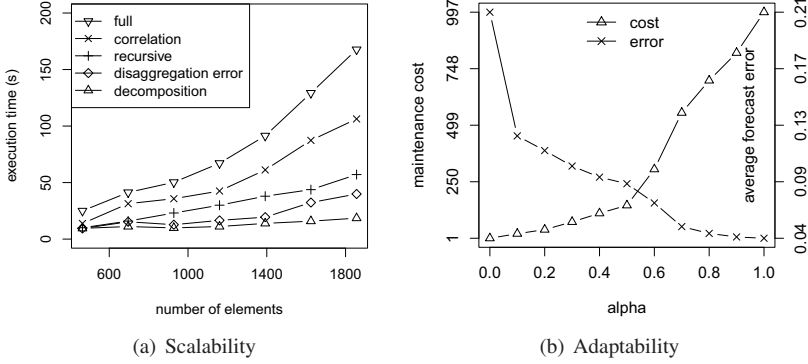


Figure 7: Scalability and Adaptability to User Requirements

series fall below the correlation threshold.

4.4 Adaptability to User Requirements

In this section, we take a closer look at the outcome of our greedy approach while varying the parameter α . Recall that an $\alpha = 0$ results in the top-down approach as this is the configuration with the minimal maintenance cost. In contrast, setting $\alpha = 1$ leads to the configuration with the best overall forecast error regardless of the maintenance cost. In this experiment, we execute our greedy approach using the whole electricity data set while increasing α from 0 to 1. Figure 7(b) presents the average forecast error and the maintenance cost of the final configuration. In the beginning, with only little additional maintenance cost we get a high improvement of the average forecast error. The reason is that our greedy approach always adds those models first that lead to the highest improvement. As α gets higher the number of additional models increases but the error improvement decreases. To conclude, with a small value of α we get the best improvement of the forecast error with low additional maintenance cost. A very high α leads to the best overall forecast error, however it might not be worse the maintenance cost. Nevertheless, for the energy data set, a value of $\alpha = 1$ leads to the creation of about 57% of the models where the average forecast error beats the bottom-up and complete approach.

In addition, we examined different kind of workloads. For this, we varied the number of distinct elements addressed by the workload, the forecast horizon and the frequency of forecast queries. The higher the number of distinct elements, the larger the search space. Therefore, with a higher number of distinct elements, we tend to a higher number of models but also to a higher accuracy as we can exploit more disaggregation and aggregation possibilities. An increase of the forecast horizon of the queries leads to a higher average error and decreasing maintenance costs for all data sets. Longer forecast horizons are harder to predict at single time series level. Due to more robustness, the greedy algorithm favors

models at higher aggregation levels leading to lower maintenance costs. Last, we varied the distribution of query frequencies by increasing the parameter z of a zipf distribution. A high parameter implies a high frequency of only a few elements in the workload (high skew) while a low parameter results in equal distribution of the frequencies (low skew). With a high skew, we strongly prefer a few workload elements leading to the creation of models favoring only those. However, the overall forecast error stays roughly constant as the error is weighted with the workload frequencies (Definition 3).

5 Related Work

Related work can be found in three main areas: (1) existing approaches to integrate forecasting in database management systems, (2) hierarchical forecasting studies in forecasting and economic literature and (3) materialized view selection.

Forecasting in DBMS Forecasting has already been successfully integrated into DBMS. For example, within the Fa system [DB07] an incremental approach is proposed to build models for a multidimensional time-series in which more attributes are added to the model in successive iterations. Furthermore, the skip-list approach for efficient forecast query processing [GZ08] proposes an I/O-conscious skip list data structure for very large time series in order to enable the determination of a suitable history length for model building. However, all approaches investigate how to efficiently find the best forecast model for one specific forecast query using database techniques. We consider the problem of efficient forecast query processing from a different point of view by addressing the interaction of queries, which allows the reduction of the forecast error and maintenance cost by reusing models. Agarwal et al. address the problem of forecasting high-dimensional data over trillions of attribute combinations [ACL⁺10]. They propose to store and forecast only a sub-set of attribute combinations and compute other combinations from those using high-dimensional attribute correlation models. However, they select the sub-set manually for historical importance and seasonality, while we propose an automatic approach to determine the optimal set of models to store. Then, their correlation models can be used as disaggregation method in our approach.

Hierarchical Forecasting In contrast to our work, hierarchical forecasting considers only a single hierarchy. In this context, the majority of the literature has focused on comparing the performance of bottom-up forecasting (individual item forecasts are made directly and aggregated) and top-down forecasting (forecasts are made at aggregate level and then allocated to individual items). Some favor the bottom-up approach [ZT00, DWD76], other the top-down approach [Fli99, NMB94] and some found no method to be superior for their specific data set [FM92]. In addition, influencing factors of the superiority of one approach over the other were investigated, e.g., quality of forecast method, correlation between variables and forecast errors [Bar80]. In a recent work, bottom-up versus top-down was investigated when the subaggregate series follows a first-order univariate moving average MA(1) process [WVP09]. They found no significant difference in the accuracy of the two strategies when the correlation between the subaggregate series is small or moderate. However, all research empirically analyzes one specific data set and conclude for one of

the two methods. In contrast, we propose an approach that quantifies different solutions and also allows for mixed solutions. In addition, existing approaches focus on accuracy only, while we take maintenance cost into account as well. This allows for faster query processing and a lower system load as maintenance time of models is reduced.

Materialized View Selection There is a high amount of work in the area of materialized view selection [ACN00, BPT97, SDN98]. For example, in [BPT97] two techniques are proposed, which reduce the number of views to consider for materialization (based on query benefit, dependent views and the size of the materialized view). The general problem is the same – for a given workload find the optimal set of materialized views with minimal query and maintenance cost. However, the model selection problem strongly differs from the materialized view selection problem, as a second dimension – the forecast accuracy – is introduced. In addition, different derivation schemes are used in model selection, i.e., disaggregation. Therefore, we cannot apply existing techniques directly.

6 Conclusion and Future Work

In this paper, we introduced the problem of physical design of time series forecast models in a multi-hierarchical data-warehouse scenario. We defined the two-dimensional optimization problem of minimizing the forecast accuracy and maintenance cost. On the one hand, we generalized the problem of finding the best hierarchical structure addressed in forecasting literature. On the other hand, we additionally include maintenance cost addressing evolving time series. Our solution consists of a greedy enumeration approach and different heuristics, which might reduce the time consumption of the offline design algorithm. In our experimental evaluation, we used three data sets to show that we can find the configuration, which reaches a high accuracy while using as less models as possible.

This paper is part of an on-going research about physical design of forecast models inside a database. We started to analyze the physical design from an offline point of view. However, an important question is how this structure should be maintained when time series characteristics change, i.e., online physical design. In detail, we need to consider three different types of maintenance. First, maintenance of the model state and disaggregation keys, which is cheap and can be done incrementally every time a new tuple arrives. Second, maintenance of model parameters, which is expensive and therefore could be triggered either time or threshold based. Third, maintenance of the model pool itself. For the last case, we need to monitor the real workload, maintenance cost and accuracy of different forecast models. Depending on these statistics, we might decide to drop a model, which has not been worthwhile or to create a new model to improve forecast accuracy.

Acknowledgment

The work presented in this paper was partially funded by the EU within the MIRACLE project under the grant agreement number 248195.

References

- [ACL⁺10] Deepak Agarwal, Datong Chen, Long-ji Lin, Jayavel Shanmugasundaram, and Erik Vee. Forecasting high-dimensional data. In *SIGMOD Conference*, 2010.
- [ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.
- [Bar80] Amir Barnea. An analysis of the usefulness of disaggregated accounting data for forecasts of corporate performance. *Decision Sciences*, 11:17–26, 1980.
- [BBD⁺10] H. Berthold, M. Böhm, L. Dannecker, F.-J. Rumph, T. B. Pedersen, C. Nychtis, H. Frey, Z. Marinsek, B. Filipic, and S. Tselepis. Exploiting renewables by request-based balancing of energy demand and supply. In *IAEE*, 2010.
- [BPT97] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized Views Selection in a Multidimensional Database. In *VLDB*, 1997.
- [Cha00] Chris Chatfield. *Time-Series Forecasting*. Chapman & Hall, 2000.
- [DB07] Songyun Duan and Shivanath Babu. Processing Forecasting Queries. In *VLDB*, 2007.
- [DWD76] D.M. Dunn, W.H. Williams, and T.L. DeChaine. Aggregate Versus Subaggregate Models in Local Area Forecasting. *Journal of the American Statistical Association*, 71:68–71, 1976.
- [Fli99] Gene Fliedner. An investigation of aggregate variable time series forecast strategies with specific subaggregate time series statistical correlation. *Computers & Operations Research*, 26:1133–1149, 1999.
- [Fli01] Gene Flidner. Hierarichal forecasting issues and use guidelines. *Industrial Management & Data Systems*, 101:5–12, 2001.
- [FM92] Eugene B. Flidner and Vincent A. Mabert. Constrained Forecasting: Some Implementation Guidelines. *Decision Sciences*, 23:1143–1161, 1992.
- [FRBL10] Ulrike Fischer, Frank Rosenthal, Matthias Boehm, and Wolfgang Lehner. Indexing Forecast Models for Matching and Maintenance. In *IDEAS*, 2010.
- [GS90] Charles W. Gross and Jeffrey E. Sohl. Disaggregation methods to expedite product line forecasting. *Journal of Forecasting*, 9:233–254, 1990.
- [GZ08] Tingjian Ge and Stan Zdonik. A skip-list approach for efficiently processing forecasting queries. *VLDB*, 2008.
- [KR02] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit*. Wiley, 2002.
- [MER10] *The MeRegio Project*, 2010. <http://www.meregio.de/en/>.
- [NMB94] S.L. Narasimhan, D.W. McLeavey, and P. Billington. *Production Planning and Inventory Control*. Allyn & Bacon, 2 edition, 1994.
- [Ora10] Oracle. Oracle OLAP DML Reference: FORECAST - DML Statement, 2010.
- [Pre10] PredictTimeSeries – Microsoft SQL Server 2008 Books Online. <http://msdn.microsoft.com/en-us/library/ms132167.aspx>, 2010.
- [SDN98] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized View Selection for Multidimensional Datasets. In *VLDB*, 1998.
- [TRA10] *Tourism Research Australia - National Visitor Survey*, 2010. <http://www.ret.gov.au/tourism/tra/domestic/national/Pages/default.aspx>.
- [US110] *US EIA - International Energy Statistics*, 2010. <http://tonto.eia.doe.gov/cfapps/ipdbproject/IEDIndex3.cfm?tid=2&pid=2&aid=2>.
- [WVP09] Handik Widiarta, S. Viswanathan, and Rajesh Piplani. Forecasting aggregate demand: An analytical evaluation of top-down versus bottom-up forecasting in a production planning framework. *International Journal of Production Economics*, 118:87–94, 2009.
- [ZT00] Arnold Zellner and Justin Tobias. A Note on Aggregation, Disaggregation and Forecasting Performance. *Journal of Forecasting*, 19:457–469, 2000.

Online Hot Spot Prediction in Road Networks

Maik Häsner Conny Junghans Christian Sengstock Michael Gertz

Institute of Computer Science
University of Heidelberg, Germany

Abstract: Advancements in GPS-technology have spurred major research and development activities for managing and analyzing large amounts of position data of mobile objects. Data mining tasks such as the discovery of movement patterns, classification and outlier detection in the context of object trajectories, and the prediction of future movement patterns have become basic tools in extracting useful information from such position data. Especially the prediction of future movement patterns of vehicles, based on historical or recent position data, plays an important role in traffic management and planning.

In this paper, we present a new approach for the online prediction of so-called *hot spots*, that is, components of a road network such as intersections that are likely to experience heavy traffic in the near future. For this, we employ an efficient path prediction model for vehicle movements that only utilizes a few recent position data. Using an aggregation model for hot spots, we show how regional information can be derived and connected substructures in a road network can be determined. Utilizing the behavior of such hot spot regions over time in terms of movement or growth, we introduce different types of hot spots and show how they can be determined online. We demonstrate the effectiveness of our approach using a real large-scale road network and different traffic simulation scenarios.

1 Introduction

Driven by major advancements in GPS-based technologies, position data of mobile objects have become ubiquitous. Through tracking systems for vehicles, persons, and even animals, enormous amounts of object trajectory data are being collected and subject to various data mining tasks. The objectives of these tasks include the discovery of (periodic) movement patterns, the classification of objects based on their trajectories, and the prediction of future movement patterns based on the objects' historical trajectories (see the tutorial by Han et al. [HLT10] for an excellent overview).

In the past couple of years, especially the analysis of traffic data has been of great interest as results obtained by analyzing past and current vehicle position data provide important input to traffic management, control, and planning. One key feature of respective approaches is the detection of *hot spots*, that is, road segments and intersections that experience a high load of traffic and thus require special attention in traffic management.

One can roughly distinguish three classes of approaches to the discovery of hot spots, based on what object position data is used and whether the detection refers to past, current, or future hot spots. The first class of approaches apply real-time monitoring where

only current vehicle position data is used to determine hot spots based on the density of vehicle positions on a given road network. This approach, where no predictions are made, is employed by several online traffic monitoring systems. The second class of approaches perform an analysis of historical trajectory data to determine (periodic) hot spots that occurred in the past and make predictions of future hot spots that likely occur, e.g., during rush hour [JYZJ10, LHLG07]. The third class utilizes only current position data, perhaps including a few previous object position data, and they try to predict near future vehicle paths and hot spots (e.g., [KRSZ08]). Especially this class is of interest in online scenarios where no information about historic object trajectories is available and near time traffic predictions have to be made to mitigate congestion in a timely manner.

In this paper, we present such an online approach for the prediction of near future traffic hot spots based on recent positions of vehicles moving in a road network. Our approach does not require historic object trajectories but makes predictions based on direction information derived from the most recent and current positions of vehicles, similar to the approach by Kriegel et al. [KRSZ08]. Through the prediction of possible future positions of vehicles, nodes in the road network (representing, e.g., intersections) are assigned a weight that reflects the likely traffic intensity within a specified time horizon. Instead of just focusing on such so-called hot spot nodes, our interest is in the discovery of regional information about hot spots, e.g., structures composed of an intersection and roads leading to that intersection. For this, we introduce a subgraph-based approach, with subgraphs being part of the underlying road network, to describe predicted hot spots in a more comprehensive and intuitive manner. We also introduce different types of hot spots based on the evolution of respective subgraph structures over time. For example, one can predict that a hot spot grows over time or that a hot spot moves in a particular direction. Respective results are obtained based on successive predictions of hot spots. We evaluate the efficiency and effectiveness of our approach to hot spot prediction using a real large-scale road network and simulated traffic scenarios. In summary, the paper makes the following contributions:

- An efficient approach to predict a vehicle's near time future positions based only on the very recent past positions.
- An online prediction approach to determine near future hot spot nodes that likely will experience heavy traffic, based on predicted future locations of vehicles.
- An aggregation model for hot spot nodes to determine regional structures in a road network and the evolution of such predicted hot spot structures over time.
- An extensive evaluation of our framework based on a real-world road network with more than 170,000 nodes and up to 10,000 vehicles in different traffic scenarios.

The rest of the paper is structured as follows. After a review of related work in the following section, in Section 3 we detail our online approach for predicting future positions of objects and how to derive hot spot nodes from such a prediction. In Section 4, we then discuss how regional patterns can be formed from hot spot nodes, and we introduce different types of hot spots based on their behavior over time. After the presentation of an

evaluation of our approach using a real-world road network and simulated traffic data in Section 5, we conclude the paper with a summary and outlook in Section 6.

2 Related Work

Research on path prediction for moving objects has been conducted for some time already, e.g., by Pelanis et al. in [PvJ06] and Tao et al. in [TFPL04]. These approaches, however, do not consider an underlying road network that restricts the possible movement of objects. More recently, approaches for path prediction of moving objects in road networks have been proposed by, e.g., Kim et al. in [KWK⁺07] and Jeung et al. in [JYZJ10]. The work in [KWK⁺07] assumes that the objects' destinations are known, which is in contrast to our approach. In addition, both approaches, and also many others that predict motion paths in road networks, utilize the complete historical trajectories of all moving objects to infer common movement or turning patterns. Our prediction instead uses a simple heuristic based on only a few of the most recent object positions to identify positions that moving objects will likely travel to in the near future, as existing path prediction approaches are prohibitively expensive and thus cannot be used in an online fashion in large-scale road networks with high numbers of moving objects.

As our focus is on the identification of *regions* with heavy traffic, our work also relates to the detection of moving clusters [KMB05], convoys [JYZ⁺08], and flocks [GvK06]. All of these moving object patterns require at least a minimum number of moving objects to be within a certain distance from each other. We use the same requirement during the detection of hot spots. The moving object patterns listed above are usually identified while, or even after, they appeared, as no methods for their prediction have been proposed, yet. However, we are able to utilize methods for analyzing the evolution of convoys, as proposed by Aung and Tan in [AT10], in order to identify different types of hot spots, like growing or moving hot spots.

The detection of hot routes and hot spots in road networks has been addressed by Li et al. in [LHLG07] and Liu et al. in [LLN⁺10] respectively. Both approaches provide an analysis of the current traffic situation, and thus do not perform a prediction but rather a detection of hot spots.

Most similar to our work is the approach by Kriegel et al. in [KRSZ08], where they introduce a method for traffic density prediction in road networks. The approach in [KRSZ08] uses a statistical traffic model to predict the traffic density of any edge in the road network for a given prediction time, which can be either a specific point in time in the future, or a time interval of certain length. Predictions done by our approach always refer to a time interval, not a specific future point in time. In addition, we predict traffic intensity on nodes instead of edges and use a different weighting function. Nevertheless, the traffic density in the road network predicted by the approach in [KRSZ08] is very similar to our concept of weights that are assigned to each node in the network to represent the predicted traffic intensity. Our approach goes one step further and considers regional aspects of hot spots by identifying hot spot regions in the form of concrete subgraphs structures in the

road network. Overall, our approach is not tailored as much to predictions for longer periods of time as [KRSZ08] is, but instead focused on an efficient and scalable approach for predicting regional hot spots and their evolution in the near future.

3 Prediction of Traffic Intensity

The prediction of hot spots requires knowledge about locations that moving objects are likely to visit within some future time window. We need to infer this information from the trajectories observed thus far, as we assume that there is no information available about future routes or destinations of moving objects. We therefore now present a method that, at a given point in time, predicts the traffic intensity at all intersections in a road network for a time interval of a certain length. Based on this prediction, locations that are likely to be visited by a high number of moving objects within the given time interval can be identified and processed for further exploration.

In the following Section 3.1, we first explain our setting, i.e., the road network and moving objects. We then detail our approach for traffic intensity prediction in Section 3.2. Methods to infer hot spots from the traffic intensity prediction are discussed in Section 3.3.

3.1 Road Network and Moving Objects

Similar to [JYZJ10], we model a *road network* as an undirected graph $G = (V, E, D)$, where V is the set of vertices, i.e., connections between two or more road segments, E is the set of edges, i.e., road segments, and D is a function that maps each edge to a set of descriptive attributes, such as the speed limit of road segments. Note that road segments could also be modeled as directed edges that indicate, e.g., one-way roads or roads where the speed limit differs for the two directions. Our approach could be easily applied to a directed graph as well.

A set of objects O is moving within the road network. Each such *moving object* $o \in O$ has an associated unique identifier and an associated trajectory $T(o)$ containing its previously observed locations. We assume a synchronized model where moving objects report their location periodically at times $t_{-n}, \dots, t_{-1}, t_0, t_1, \dots$, with $n \in \mathbb{N}$, where t_0 denotes the current point in time, times with negative subscript are in the past, and times with positive subscript are in the future. We denote each such time t_i a *time step*. The duration between two consecutive time steps t_i and t_{i+1} is called the *step size*, which is application specific and can range from fractions of a second to minutes, or even hours, although the latter is uncommon in road network applications. When a moving object reports its location at time t_i , it is of the form of a *measurement* $l_i(o) = (\mathbf{x}_i(o), v_i(o), t_i)$, where $\mathbf{x}_i(o)$ is o 's position and $v_i(o)$ is its velocity. An object's position is usually a 2-dimensional vector containing longitude and latitude information, and we assume it to be on an edge or vertex in the road network.

Overall, the trajectory of an object $o \in O$ is an ordered list of measurements, i.e., $T(o) =$

$\langle (\mathbf{x}_{-n}(o), v_{-n}(o), t_{-n}), \dots, (\mathbf{x}_{-1}(o), v_{-1}(o), t_{-1}), (\mathbf{x}_0(o), v_0(o), t_0) \rangle$, where t_{-n} is the oldest time step and thus represents the first measurement reported by object o . For our approach, it is not necessary to store the entire trajectory $T(o)$ of an object o . The two most recent measurements of each object are sufficient for the OPS approach. Note that a trajectory contains measurements up to the current time step t_0 , but no information about the object's future positions. Predicting likely positions of an object within a certain time horizon in the future is part of our proposed method.

In a practical setting, the trajectories our approach uses for hot spot prediction represent a sample of all cars moving in the road network at any given time. As we will demonstrate in the evaluation in Section 5, a sample of all moving objects yields similar prediction results as the full set of moving objects would.

3.2 The OPS Approach

We now present *OPS*, an approach for Online Prediction of Hot SpotS. Given trajectories $T(o)$ for all objects $o \in O$ at the current point in time t_0 , our goal is to predict the nodes in the road network that these objects will likely visit within the next h time steps, i.e., in the time interval $[t_0, t_h]$. We predict these nodes separately for each object and denote the predicted set of nodes at time t_0 as $A_0(o)$. Note that each $A_0(o)$ is a subset of all vertices in the road network, thus $A_0(o) \subseteq V$, and $A_0(o)$ may be different for each object. In addition, a weight $w_0(v, o) \in [0, 1]$ needs to be determined for each node $v \in A_0(o)$ that indicates how likely it is for o to visit v . These weights are an essential aspect of the OPS approach, as nodes in $A_0(o)$ should not all contribute equally to the traffic intensity prediction, but based on how likely they actually are part of the future path of a moving object. By aggregating the weights $w_0(v, o)$ that all objects $o \in O$ contribute to a node v , the overall weight $w(v)$ for v is known, which is an estimator for its traffic intensity within the next h time steps.

A *hot spot* that is predicted at time t_0 is a part of the road network where a significant number of objects $o \in O$ are likely to be located in the time interval $[t_0, t_h]$. A *hot spot node* is therefore defined as a node $v \in V$ having a weight $w(v)$ that is relatively high with respect to the weights of most other nodes. We infer hot spot nodes based on the weights $w(v)$ as a last step in the presented approach (cf. Section 3.3).

In the next paragraphs, we will detail individual aspects of the algorithm, i.e., assigning and updating the weights $w(v)$ at each time step and extracting hot spot nodes based on the weights of all nodes. The complete OPS approach is given in Algorithm 1.

Weight assignment For each object o , a weight $w_0(v, o)$ that is greater than zero is only assigned to vertices that are likely to be visited by o within the next h time steps. In order to determine the subset of nodes that might be affected, i.e., the set $A_0(o) \subseteq V$, we first perform a coarse prediction of o 's path during the next h time steps as follows (cf. lines 9–14 of Algorithm 1).

Let $\mathbf{d}_0(o) = \mathbf{x}_0(o) - \mathbf{x}_{-1}(o)$ be o 's direction of movement at time t_0 . Then, $\mathbf{x}_h(o) =$

Input: graph $G = (V, E, D)$, trajectories $T(o)$ for each $o \in O$, time horizon h
Output: set $hn(t_0)$ of all hot spot nodes predicted at the current time t_0

```

1 foreach  $v \in V$  do // initialization
2    $w(v) = 0$ ;
3    $c(v) = \emptyset$ ;
4 foreach new time step do
5   foreach moving object  $o \in O$  do
6     foreach  $v \in A_{-1}(o)$  where  $\mathbf{d}_0(o) \cdot (\mathbf{x}(v) - \mathbf{x}_0(o)) < 0$  do // remove
       weights
7        $w(v) = c(v)[o]$ ;
8       remove  $o$  from the set  $c(v)$ 
       // compute new  $A_0(o)$ 
9        $\mathbf{x}_h(o) = \mathbf{x}_0(o) + h \cdot v_0(o) \cdot (\mathbf{x}_0(o) - \mathbf{x}_{-1}(o))$ ;
10       $\mathbf{x}'_h(o) = \text{node } v \in V \text{ closest to } \mathbf{x}_h(o)$ ;
11       $\mathbf{x}'_0(o) = \text{node } v \in V \text{ closest to } \mathbf{x}_0(o) \text{ in direction of } o\text{'s movement}$ ;
12       $A_0(o) = \text{all nodes visited by } A^*(\mathbf{x}'_0(o), \mathbf{x}'_h(o))$ ;
13      foreach  $v \in A_0(o)$  do // assign and update weights
14         $w_0(v, o) = 1 - (\text{cost}(\mathbf{x}'_0(o), v) / \text{cost}(\mathbf{x}'_0(o), \mathbf{x}'_h(o)))$ ;
15         $w(v) += w_0(v, o)$ ;
16         $c(v)[o] += w_0(v, o)$ ;
17    mean = mean of weights  $w(v)$ ,  $\forall v \in V$ ;
18    std = standard deviation of weights  $w(v)$ ,  $\forall v \in V$ ;
19    foreach  $v \in V$  do
20      if  $w(v) \geq \text{mean} + 3 * \text{std}$  then
21        add  $v$  to the set  $hn(t_0)$ ;
22  return  $hn(t_0)$ 

```

Algorithm 1: OPS approach to determine hot spot nodes in the road network G

$\mathbf{x}_0(o) + h \cdot v_0(o) \cdot \mathbf{d}_0(o)$ is the point in space that object o travels to within the next h time steps, assuming o maintains its current direction of movement and velocity. As $\mathbf{x}_h(o)$ is not necessarily on the road network G , we map it to the closest vertex in G , denoted $\mathbf{x}'_h(o)$, by performing a nearest neighbor search. As we only consider vertices for our traffic intensity prediction, we also need to map o 's current location to the closest node $\mathbf{x}'_0(o)$ in G , which we do by finding the vertex that is closest to $\mathbf{x}_0(o)$ towards $\mathbf{d}_0(o)$. Figure 1 depicts an example scenario. The reported locations of o at time steps t_{-1} and t_0 are marked by blue rectangles, as is the predicted location at time t_h . In addition, nodes $\mathbf{x}'_0(o)$ and $\mathbf{x}'_h(o)$, i.e., the vertices in the road network that $\mathbf{x}_0(o)$ and $\mathbf{x}_h(o)$ were mapped to, are marked by green triangles. Object o 's direction of movement $\mathbf{d}_0(o)$ is depicted as a blue arrow. Note that there are other options to compute $\mathbf{d}_0(o)$, e.g., as the average direction of the k most recent time steps.

Given $\mathbf{x}'_0(o)$ and $\mathbf{x}'_h(o)$, i.e., the start and end nodes of o 's movement during the next h time steps, we find the set $A_0(o)$ of likely visited vertices using the routing algorithm

A^* [HNR68]. A^* performs a best-first search to find the lowest-cost route from $\mathbf{x}'_0(o)$ to $\mathbf{x}'_h(o)$. To compute the cost $\text{cost}(v_1, v_2)$ for the route between two nodes v_1 and v_2 , function D of the graph is used to determine the speed limit of each road segment. Thus, the lowest-cost route found by the A^* algorithm is the *fastest* route. A^* uses heuristics to cut off the search on paths that are not likely to contribute to the fastest route. Thus, all nodes visited by A^* are in the vicinity of the fastest route and thus likely to be visited by o during the time interval $[t_0, t_h]$. Hence, all nodes visited by A^* during the computation of the fastest route from $\mathbf{x}'_0(o)$ to $\mathbf{x}'_h(o)$ are in the set $A_0(o)$, and thus, $\forall v \in A_0(o): w_0(v, o) > 0$. In Figure 1, these nodes are marked by filled black circles.

The value of each weight $w_0(v, o)$ at time t_0 should indicate how likely o is to visit v within the next h time steps. Nodes that can be reached from the known start node $\mathbf{x}'_0(o)$ within a short amount of time are more likely to be visited, because the object o may change direction or velocity at some point in time $t_i < t_h$ and thus our predictions for likely visited nodes become less accurate as the time traveled by o since t_0 increases. Nodes in $A_0(o)$ that can be reached faster by o are therefore assigned a higher weight than those where o takes longer to reach them. We compute the weight $w_0(v, o)$ of each node $v \in A_0(o)$ as follows:

$$w_0(v, o) = 1 - \frac{\text{cost}(\mathbf{x}'_0(o), v)}{\text{cost}(\mathbf{x}'_0(o), \mathbf{x}'_h(o))} \quad (1)$$

That is, the assigned weight $w_0(v, o)$ decreases as the A^* -based cost between the start node and v increases. Note that $\forall v \in A_0(o) \setminus \mathbf{x}'_h(o): \text{cost}(\mathbf{x}'_0(o), v) < \text{cost}(\mathbf{x}'_0(o), \mathbf{x}'_h(o))$, as A^* performs a best-first search and thus no node in $A_0(o)$ can have higher cost with respect to start node $\mathbf{x}'_0(o)$ than node $\mathbf{x}'_h(o)$. Therefore, $w_0(v, o) \in [0, 1]$ always holds. In the example in Figure 1, nodes v_1 and v_2 are two of the nodes that are assigned a weight by object o at that time. The weight assigned to v_1 is higher than that for v_2 , i.e., $w_0(v_1, o) > w_0(v_2, o)$.

Weight update Weight assignment is performed at every time step right after new measurements arrive from all moving objects. It is therefore necessary to update the weights of all nodes $v \in V$, specifically to aggregate weights $w_i(v, o)$ that are assigned at different time steps, and to reweight weights that were contributed by objects that, at time t_0 , are no longer likely to visit v (cf. lines 6–8 and 15–16 of Algorithm 1).

At every point in time, the weight $w(v)$ of a node v is the sum of all weights $w_i(v, o)$, i.e., the weights that all objects have contributed for node v . We use the set $c(v)$ to keep track of all objects that contributed to $w(v)$. If an object o repeatedly contributes a weight greater than zero to node v , i.e., $v \in A_i(o)$ holds, in consecutive time steps t_i , these weights are added up instead of just counting the most recent weight that o contributed to v . This is because summing up the assigned weights over several consecutive time steps provides for a traffic intensity prediction that spans more than one time step, and thus helps to identify “popular” nodes. This aspect of the OPS approach is in contrast to most existing approaches, which mostly operate on snapshots and do not take the previously discovered popularity of locations into account. That is, other approaches start over every

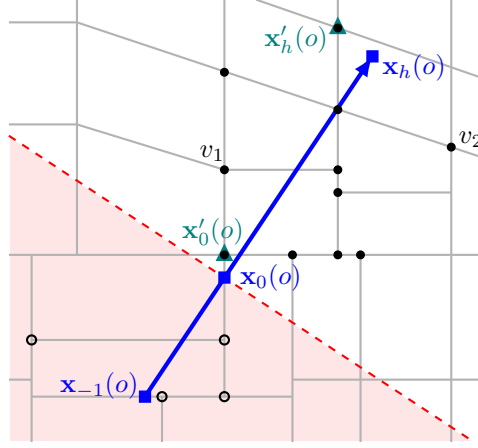


Figure 1: Example scenario depicting weight assignment, update, and revocation in OPS

time a prediction is done and thus disregard previously obtained information about the popularity of locations, whereas OPS maintains this information and uses it to reinforce the expressiveness of the prediction. This is because if an object o repeatedly assigns weights greater than zero to a node v in consecutive time steps, it indicates that it becomes more and more likely for o to visit v .

However, adding up all weights continuously does not yield an accurate prediction for the *future* traffic intensity, as the sum of all weights contributing to a node's overall weight $w(v)$ simply represents v 's all-time likelihood to be visited by moving objects. We therefore revoke weights contributed by certain objects, to account for the event that an object o is not likely any more to visit v . Specifically, at time t_0 , all weights $w_i(v, o)$ that were contributed by nodes $v \notin A_0(o)$ at time steps $t_i < t_0$ should be subtracted from $w(v)$. Note that we store the sum of all weights contributed to a node v by object o in $c(v)[o]$. Consider an example where an object o contributed the following weights to node v : $w_{-3}(v, o) = 0$, $w_{-2}(v, o) = .35$, $w_{-1}(v, o) = 0.2$, and $w_0(v, o) = 0$. Thus, at t_0 , a total weight of $.35 + 0.2 = 0.55$ should be subtracted from $w(v)$.

Revoking weights in this fashion requires to determine for each object o all nodes v for which $v \notin A_0(o)$ and $v \in A_{-1}(o)$ hold, i.e., nodes that are not an element of $A_0(o)$ at time t_0 , but have been an element of $A_{-1}(o)$ at time t_{-1} . In order to do this, one would have to store $A_{-1}(o)$ and compute the set difference $A_{-1}(o) \setminus A_0(o)$ for all objects o , which, for large graphs, requires a lot of additional memory and is computationally expensive. Therefore, we instead use an efficient heuristic to determine nodes where weights need to be revoked. The basic idea is depicted in Figure 1. The dashed red line is perpendicular to object o 's direction of movement $\mathbf{d}_0(o)$ and crosses at o 's current location $x_0(o)$. All nodes v located in the shaded area below the dashed line have already been "passed" by object o and are thus not likely to be visited by o in the (near) future. Specifically, object o either did visit the nodes in the shaded area already or it could have visited them using a shorter route before t_0 , as visiting any of these nodes now, i.e., after time t_0 , would require

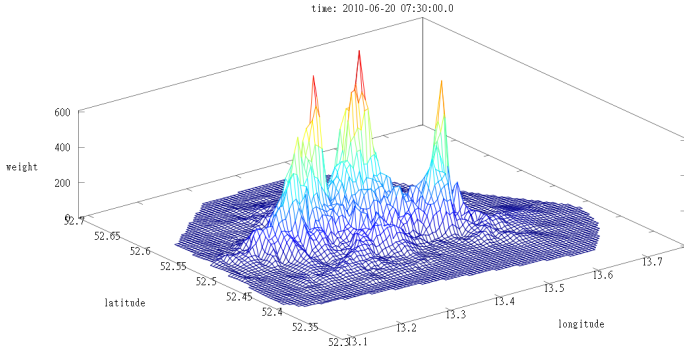


Figure 2: Traffic intensity prediction for an example data set at time $t_0 = 7:30\text{am}$ on June 20th, 2010, using $h = 120$

o to take a detour, which is much less likely.

For nodes v that are located in the shaded area, the following inequality holds:

$$\frac{\mathbf{d}_0(o) \cdot (\mathbf{x}(v) - \mathbf{x}_0(o))}{|\mathbf{d}_0(o)| \times |\mathbf{x}(v) - \mathbf{x}_0(o)|} < 0 \Leftrightarrow \mathbf{d}_0(o) \cdot (\mathbf{x}(v) - \mathbf{x}_0(o)) < 0 \quad (2)$$

Thus, at time t_0 , we revoke weights from all nodes $v \in A_{-1}(o)$ for which inequality 2 holds. Such nodes are marked by empty circles in the example in Figure 1. Note that $\mathbf{x}(v)$ in the above inequality denotes the position of vertex v . Checking if weights need to be revoked can be done at the beginning of each time step, i.e., before the new set $A_0(o)$ is computed for all moving objects. Then, this step does not require any additional memory, as $A_{-1}(o)$ is still present at the beginning of a time step, and can be overwritten by $A_0(o)$ once all nodes $v \in A_{-1}(o)$ have been checked according to inequality 2.

3.3 Identification of Hot Spot Nodes

After updating the weights $w(v)$ of all nodes $v \in V$ in the road network at time t_0 , one needs to analyze the distribution of weights over the road network in order to identify hot spot nodes. Figure 2 depicts the weights assigned to all nodes in a given road network at a specific point in time $t_0 = 7:30\text{am}$ based on an example data set of trajectory data. As is obvious from the figure, it is easy to identify areas in the road network where an exceptionally high traffic intensity is predicted for time interval $[t_0, t_h]$ ($h = 120$ in the figure). Vertices having a high weight stand out as peaks, and thus one can identify a total of three hot spots in the example in Figure 2.

While the visual identification of hot spots is straightforward, a more formal definition of hot spots is needed in order to automatically extract them from the set of all nodes in G . A hot spot is comprised of a single vertex, i.e., it is a hot spot node, or a set of connected vertices in the road network. The latter group of hot spots is discussed in Section 4, as

several different types of such hot spots can be distinguished. In the following, we show how to identify hot spot nodes based on all weights $w(v)$.

Definition 3.1 (Hot Spot Nodes) *A hot spot node predicted at time t_0 is a vertex $v \in V$ that fulfills the following two properties:*

1. *The weight $w(v)$ of node v is significantly higher than the weight of most other vertices in the road network, based on some measure.*
2. *At least $\min MO$ distinct moving objects $o \in O$ contributed to v 's weight $w(v)$, i.e., $|c(v)| \geq \min MO$ holds.*

We refer to the set of all hot spot nodes predicted at time t_0 as $hn(t_0)$.

The second property above ensures that a hot spot node is indeed likely to be visited by a sufficiently large number of moving objects. This is very similar to the detection of moving clusters, where a minimum number of objects is required in order to constitute a cluster. Regarding property (1), different methods can be applied to identify high weights, such as a threshold δ , which may be fixed or adjustable based on network density and number of present moving objects, or outlier detection approaches, which are perfectly suited to detect weights that are significantly higher than those of most other vertices.

In the OPS approach, we use an efficient outlier detection method from [FPP97] that labels all nodes v as outliers having a weight $w(v)$ that is more than three standard deviations above the mean weight of all nodes $v \in V$ (cf. lines 17–22 of Algorithm 1). All such nodes are added to the set $hn(t_0)$ of hot spot nodes at time t_0 . Of course, depending on the desired application and the computational resources available, other outlier detection approaches may be used. However, the basic method used in the OPS approach proved to be sufficient, as we demonstrate in the evaluation in Section 5.

The set $hn(t_0)$ of hot spot nodes is updated at every time step to reflect the traffic intensity predicted in the road network at time t_0 . That is, each node $v \in hn(t_0)$ is predicted to experience an amount of traffic in the time interval $[t_0, t_h]$ that is significantly larger than the amount of traffic that will be observed at most other nodes in the road network, and thus we expect the number of moving objects visiting v to be very high compared to the number of objects visiting other nodes in the road network.

4 Regional Hot Spots and Hot Spot Patterns

Although hot spot nodes already provide some interesting information about parts of the road network, e.g., intersections that likely will experience high traffic in the near future, there can be many such nodes at any given time. Relationships among hot spot nodes and in particular their aggregated future evolution is not that obvious. To address these issues, in the following section, we present a simple clustering approach for hot spot nodes to

determine more meaningful subgraph structures that convey regional information. Using such subgraphs, in Section 4.2, we then introduce different types of predicted hot spot patterns in a road network in terms of their evolution over time.

4.1 Regional Hot Spots

Given a set of hot spot nodes $hn(t)$ determined at time t , which typically corresponds to the current time t_0 . To provide the user an aggregated view on these nodes and to better determine structures and their evolution in future predictions, a set of *subgraphs*, denoted $hg(t)$ is derived from $hn(t)$ as follows:

- each node $v \in hn(t)$ belongs to exactly one subgraph $g \in hg(t)$, and
- if two nodes $v, v' \in hn(t)$ are connected in the road network, i.e., $(v, v') \in E$, then v and v' belong to the same subgraph $g \in hg(t)$.

Thus, $hg(t)$ is the set of connected subgraphs based on hot spot nodes $hn(t)$, and the subgraphs are naturally embedded in the road network (see Figure 3 for an example). Subgraphs formed in this way naturally convey regional aspects of predicted hot spots. Such a regional view, as part of the underlying road network, can be extended further by allowing the formation of larger subgraph structures and thus regions. For example, two subgraphs $g, g' \in hg(t)$ can be combined into a single subgraph if there exists a node $v \notin hn(t)$ and g, g' are connected via v , as illustrated in Figure 3(c).

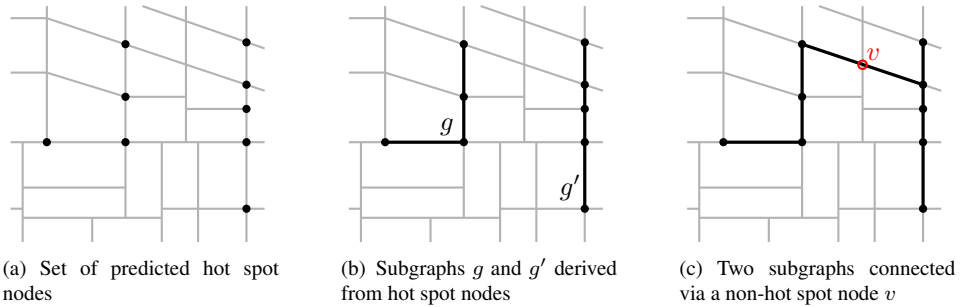


Figure 3: Subgraphs (b. and c.) derived from hot spot nodes (a.) in a road network

Recall that with each node in a subgraph $g \in hg(t)$ a weight above a certain threshold is associated (cf. Section 3.3). Using an appropriate visualization, e.g., such as the one shown in Figure 2, interesting patterns can emerge from subgraph structures. For example, a subgraph may represent an intersection (with a predicted high traffic) with road segments leading to that intersection node and respective nodes of the road segments having a decreasing weight the farther away they are from the intersection. The question that remains, however, is how such patterns predicted at a point in time t now evolve over time,

i.e., in subsequent predictions. This is an important aspect supported by our framework and is discussed in the following section.

4.2 Hot Spot Pattern

A set of subgraphs representing regional aspects of predicted hot spots is determined at each time step. Because of the continuous movement of objects in the road network, new subgraphs can emerge, existing ones remain for a period of time or they evolve in terms of expansion, shrinking, or movement. For example, a subgraph representing roads (or road segments) leading to an intersection may grow over time, reaching some kind of peak, and then shrinks until it disappears in the prediction of future hot spots. This is a typical example of an evolving pattern that emerges, e.g., during rush hour.

In general, for decision purposes in traffic planing and management, it is important to study the evolution of predicted regional hot spots over time, an important aspect not covered in related work.

In the following, we introduce a framework to determine important hot spot patterns that occur over time. For this, we assume a sequence of sets of subgraphs $L = [hg(t_{-k}), hg(t_{-k+1}), \dots, hg(t_0)]$, $k \geq 2$, which has been determined up to the current time t_0 . We thus investigate hot spot patterns that occur in a time interval of length $k + 1$. Clearly, the larger the interval, the more (stable) patterns can be determined. Typically, the size of the interval is chosen as a multiple of the step size in the trajectory data.

The most simple type of a hot spot pattern is as follows:

Definition 4.1 (Strict Stationary Hot Spot) *Let g be a subgraph of some set of subgraphs $hg(t) \in L$. g is a strict stationary hot spot if for all $hg(t') \in L$, $g \in hg(t')$.*

In other words, the exact same subgraph has to appear at each point in time in the given interval. This property is independent of possible variances of the weight of the nodes in g for this time interval. An example of such a pattern is a shopping center or large parking lot where vehicles stay for some time. Such type of patterns might be rare and perhaps less interesting in a real world setting, as hot spots and subgraph structures, respectively, are more likely to expand or shrink over time.

Definition 4.2 (Growing stationary hot spot) *Let g be a subgraph of $hg(t_{-k})$. g is said to be a growing stationary hot spot if*

1. *there exists a subgraph g_i in each $hg(t_i)$, $-k \leq i \leq 0$, s.t. g is a subgraph of g_i , and*
2. *for each two such consecutive subgraphs g_i, g_{i+1} from $hg(t_i)$ and $hg(t_{i+1})$, respectively, we have that g_i is a subgraph of g_{i+1} .*

The concept of a shrinking stationary hot spot can be formulated in an analogous way, here with g being a shrunken subgraph observed at time t_0 . These two types of hot spot

patterns can also be combined to form a new pattern, first growing, then shrinking, in the given interval. Such a pattern would then formalize the scenario we gave above describing the evolution of traffic centered around an intersection.

The final structural pattern we introduce here are *moving hot spots*. Their concept is motivated by the following scenario. Assume a truck on a highway that cannot be passed by other cars. Such a scenario relates to the leadership pattern described by Andersson et al. in [AGLW08]. Over time, more and more cars will drive behind that truck that is driving towards its destination, thus forming a hot spot that moves within the road network. A precise formal definition of such a pattern, which we are only able to discover in a very restricted setting in our system, is a little bit more involved as several parameters can be introduced to tailor the properties of such a pattern. But, in general, a subgraph $g \in hg(t_{-k})$ representing such a hot spot is moving if at least l nodes of g can be found in a subsequent subgraph $g' \in hg(t_{-k+1})$, and again for this subgraph g' at least l nodes can be found in a subsequent subgraph $g'' \in hg(t_{-k+2})$, and so on. Parameter to this pattern then is the number of nodes that must be shared among subgraphs in consecutive timestamps.

The patterns of predicted hot spots described thus far give a fairly comprehensive regional view of patterns in terms of parts of a given road network but they only consider structural properties for a given (sliding) time interval. Recall that at each time step, in addition to the weight of a node, we also maintain the set of objects that contributed to that weight (cf. Section 3.2). This information can be explored as well in the context of the above pattern. Here we only outline the basic idea. An example can be best illustrated in the context of a moving hot spot. The leadership-like pattern mentioned above has the property that mostly the same cars contribute to the weight of the nodes forming the pattern, as cars cannot pass the truck. An example where the moving objects contributing to the hot spot nodes of a pattern is not that homogeneous would be the case for the intersection. As cars pass the intersection and new cars approach the intersection, the set of moving objects contributing to respective hot spot nodes would not be as homogeneous as in the leadership pattern.

We finally want to address the computational aspects regarding the discovery of predicted hot spot patterns. Key to the complexity are (1) the size $k + 1$ of the interval (number of time steps, respectively) in which hot spot patterns are to be discovered and (2) the number of subgraphs that have been determined for each element in the sequence $L = [hg(t_{-k}), hg(t_{-k+1}), \dots, hg(t_0)]$. Let L' denote the set of subgraphs determined at time point t_1 , which comprises $[hg(t_{-n+1}), \dots, hg(t_0), hg(t_1)]$. If there is a subgraph g corresponding to a stationary, growing, or moving hot spot in L , then it only has to be checked whether there is a subgraph g' in $hg(t_1)$ that is a respective continuation of g . In contrast to this simple case, the more complex case for which all subgraphs in L' have to be processed is when there now is a subgraph in $hg(t_1)$ such that a new stationary, growing, or moving hot spot can be formed based on the subgraphs in L' . As we will show in the following section for different traffic scenarios, the total number of subgraph structures determined at each point in time is quite moderate, compared to the total number of hot spot nodes. Thus, processing respective subgraph structures for an interval of length $k + 1$ only requires minimal overhead. A key requirement, of course, is that based on the step

size vehicle positions are obtained, the computation of hot spot nodes and structures at each time step has to be done within a two steps sizes. That this requirement is satisfied will be demonstrated in the following section.

5 Experimental Evaluation

We now present some experimental results evaluating the efficiency of the OPS approach as well as properties of the predicted hot spot nodes and subgraph structures. As we use synthetic data in the evaluation, we first provide details about the generated test data in Section 5.1. We then present evaluation results regarding efficiency and the predicted hot spots in Sections 5.2 and 5.3 respectively.

5.1 Test Data and Experimental Setup

We use synthetic trajectory data to evaluate the OPS approach, as this enables us to generate data in different scenarios. Trajectories were created using our moving object simulator called *Tragor* (Trajectory generator on road networks), which simulates the movement of cars on a world-wide road network obtained from OpenStreetMap [OSM]. In Tragor, different types of cars exist. Two of them are used in our experiments: *destination cars* drive along the fastest route from a pre-defined start position to a pre-defined destination. In contrast, *random cars* follow a random path by starting at a random position in the road network and taking random turns at each intersection. Cars in Tragor always obey the speed limit, which is provided as a property of each road segment in the OpenStreetMap data set. We chose to use Tragor because it enables us to include different kinds of hot spot patterns in our test data sets.

For each of our experiments, we generate sets of trajectories that contain the movement of n_d destination cars and n_r random cars. Thus, each set has a total of $n = n_r + n_d$ moving object trajectories. We create hot spots by assigning each destination car one of e distinct destinations, $e \ll n_d$, such that the same number of destination cars is assigned to each of the e destinations. Random cars are added as noise, since they will not contribute to any hot spots due to their random movements.

For the simulation, we do not use the entire road network, but choose two different regions of interest. The first region is the city of Berlin, Germany, which represents a dense road network, and the second one is the German State of Brandenburg, which is much more sparse than the Berlin road network. Thus, when generating trajectory data sets, the moving objects are either moving on the Berlin road network, denoted *BLN*, or on the Brandenburg road network, denoted *BB*. The BLN (BB) road network contains about 180,000 (408,000) nodes and covers roughly 340 (11,380) square miles. We will indicate the road network used in the description of the experiments below.

All experiments were conducted on a 16 core Intel(R) Xeon(R)E5520 @ 2.27GHz with 48GB of main memory running Debian Linux. The OPS approach was implemented in

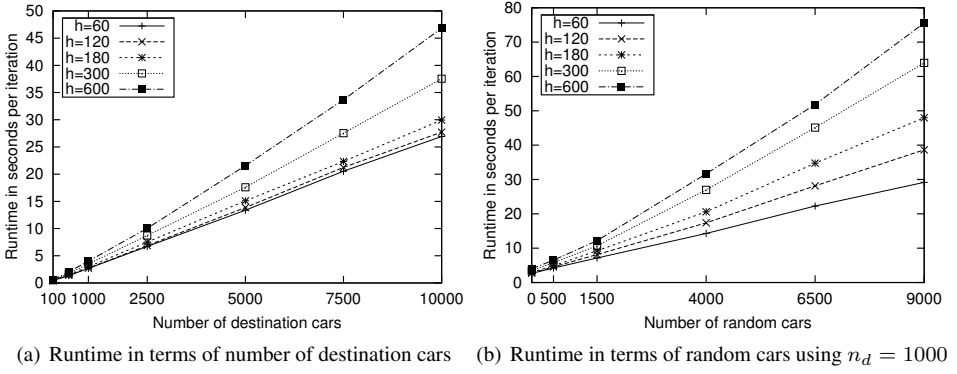


Figure 4: Runtime per iteration of OPS approach in BLN road network

Java.

5.2 Efficiency of OPS

In a first set of experiments, we evaluated the runtime of the OPS algorithm using 14 different sets of trajectories, seven each for the BLN and the BB road network. The test data sets contain the trajectories of between $n = 100$ and $n = 10,000$ moving objects, all of them destination cars that drive to one of $e = 10$ pre-defined destinations. Trajectories were generated for a period of one hour using a step size of 60 seconds, resulting in a total of 60 measurements for each moving object. Hot spot prediction was done for five different time horizons between 1 and 10 minutes. The results of these experiments are shown in Figure 4(a) for the BLN road network.

Runtimes depicted in Figure 4(a) are the total time for processing all new measurements in one time step, i.e., the runtime required for one iteration of the OPS approach. As can be seen in the figure, the runtime of OPS grows linearly with the number of moving objects, and is only slightly affected by the length of the prediction interval, i.e., the time horizon h . Specifically, for $h = 60$ the runtime is 27 seconds, while the runtime for a ten-times larger time horizon of $h = 600$ seconds has not even doubled, being less than 47 seconds. This demonstrates that our approach is very scalable both in terms of the number of moving objects and the length h of the prediction interval.

The efficiency of our approach is mostly due to the use of the A^* algorithm for predicting likely visited nodes and our heuristic for revoking weights. Only nodes that are relevant for the hot spot prediction are considered during weight assignment and update, and thus the total number of weights maintained in the graph over time is small, yielding a low runtime. To evaluate this aspect further, we performed a similar set of experiments with test data sets where we added random cars for noise. Specifically, we used $n_d = 1000$ destination cars and added between $n_r = 500$ and 9000 random cars, resulting in a total number of moving objects of up to 10,000, i.e., $n = n_d + n_r = 1000 + 9000 = 10,000$.

The results of this set of experiments are shown in Figure 4(b).

The increase of the runtime is similar to the one observed in the experiments using just destination cars. However, two differences can be observed: (1) the individual runtimes are higher than those in the respective previous experiments, and (2) the increase in runtime becomes steeper as the ratio of random cars to destination cars increases. Both observations are caused by the random movement of the majority of the cars, i.e., all random cars. Due to their random movement, each car o has very different sets $A_i(o)$ of likely visited nodes at each time step t_i , resulting in more computational effort to maintain the weights assigned to all nodes. This is because for a random car, a higher number of distinct nodes get assigned weights, whereas destination cars tend to assign weights to a similar set of nodes at each time step, resulting in a lower number of weights to be maintained overall. We will demonstrate this fact in further experiments below. Regarding the second observation above, the additional computational effort due to the random cars becomes more predominant when the ratio of random cars and destination cars increases, as is the case in our experiments, where $n_d = 1000$ is fixed and the number n_r of random cars increases.

Both observations above support our claim that OPS is efficient because of its ability to focus on nodes that are relevant to the hot spot prediction. Random cars interfere with this approach by adding irrelevant weights and nodes, thus reducing the efficiency of OPS. Runtimes for real-world trajectory data will usually be more similar to the ones shown in Figure 4(a), as real-world cars rarely exhibit random movement patterns, but instead drive fairly targeted to their destination.

We performed the same experiments as above, i.e., using just destination cars and using an increasing number of random cars, for trajectories on the BB road network, and observed runtimes that were 10 to 15 times faster than those for the BLN road network. This is because the BB road network is much more sparse, resulting in a much lower number of nodes being visited by the A^* algorithm, and thus less computational effort overall.

To further explore the efficiency of OPS in terms of the number of weights that are assigned to nodes, we collected additional measurements during the experiments described above. Each node v in the road network keeps track of the moving objects that contributed to v 's weight $w(v)$. By summing up, over all nodes, the number of objects that contributed to the node's weight, we gain insights in how the number of destination cars and random cars affects the distribution of weights in the road network. For this, we determined at the end of each iteration of OPS the sum of how many distinct moving objects contributed to the weight of each node. That is, we summed up the number of objects stored in the set $c(v)$, i.e., $|c(v)|$, over all $v \in V$. We denote this sum as $\sum(|c(v)|)$. In the following Figure 5, the maximum value of $\sum(|c(v)|)$ out of all 60 iterations is given.

In Figure 5(a) it can be seen that for higher numbers n_d of destinations cars and longer prediction horizons h the sum of assigned weights increases. This is expected, as for higher values of h a car is able to visit more nodes during the prediction time interval. Additionally, we can observe in Figure 5(b) that for high numbers of random cars, the number of assigned weights is 10 to 20 times higher than in a scenario with an equal number of destination cars. This again supports our claim that the weight assignment performed by destination cars, which behave very similar to cars in a real-world setting, is

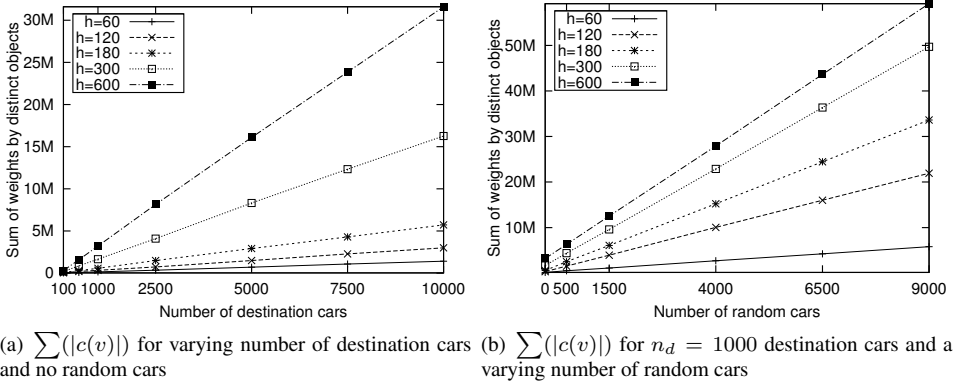


Figure 5: The number of weights assigned to nodes by distinct objects was determined for different numbers of moving objects n_d and n_r

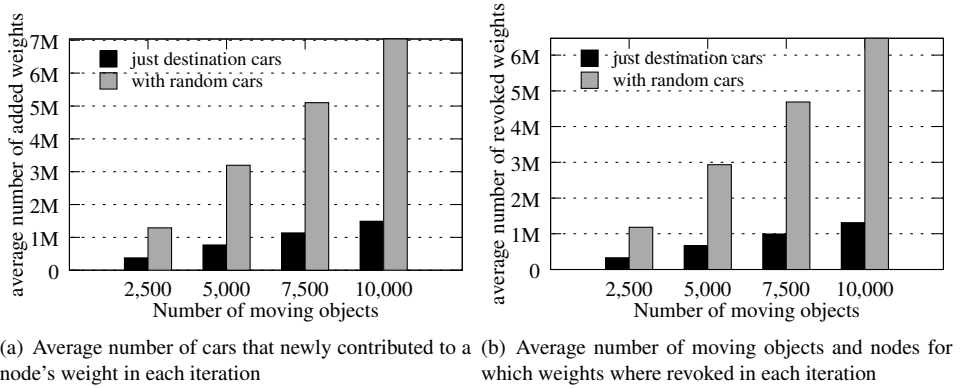


Figure 6: Average number of weights that were added or revoked in each iteration of OPS

much more focused, yielding expressive weights at each node in the road network. Also, the much higher number of assigned weights in the presence of random cars justifies the increased runtime observed in the second set of experiments, whose results were depicted in Figure 4(b).

We also kept track of how many objects newly contributed to a node's weight in each iteration and for how many objects weights were revoked from nodes. We again summed up these values over all nodes. The results are depicted in Figure 6, and show the average over all 60 iterations. These results are in line with all previous experiments, confirming that random cars indeed exhibit a significant fluctuation in terms of the distinct nodes to which they assign weights. Both Figures 6(a) and 6(b) show that in the scenarios with a significant number of random cars, a much higher number of weights gets added and revoked in each iteration, compared to scenarios where only destination cars move in the road network.

n	t_{10}		t_{20}		t_{30}		t_{40}	
	$ hn $	$ hg $	$ hn $	$ hg $	$ hn $	$ hg $	$ hn $	$ hg $
1000	319	51	383	52	526	56	638	50
2000	328	50	478	52	571	61	660	54
3000	358	46	502	48	625	52	683	49

Table 1: Number of predicted hot spot nodes and subgraphs using $\varepsilon = 1$ and minPts = 3

5.3 Evaluating Predicted Hot Spots

Next, we performed a set of experiments to evaluate the predicted hot spot nodes and subgraphs. For this, we used test data sets containing $n = n_d = 1000, 2000$, or 3000 destination cars that are driving to $e = 10$ distinct destinations in the BLN road network. It is important to note that we derived the data sets with fewer objects from the data set containing all 3000 moving objects, and thus, the hot spots should be similar throughout all three data set. We again used a step size of 60 seconds and simulated traffic for one hour, resulting in 60 measurements in each trajectory. All cars start at different positions, but as each 10% of the present cars share a destination, we expect them to “meet” at major roads and follow the same few access roads to any of the given destinations. In our experiments, we determined hot spot nodes after 10, 20, 30, and 40 time steps using a time horizon of $h = 120$ seconds. From the set of hot spot nodes, we derived according subgraphs as detailed in Section 4.1 and depicted in Figure 3(c). That is, we consider two hot spot nodes to belong to the same subgraph if they are connected directly or via at most one non-hot spot node.

We used the well-known clustering algorithm DBSCAN [EKSX96] to derive the subgraphs. The distance measure we used is the number of “hops” in the road network, i.e., the number of edges one has to use in order to get from one node to another. Thus, using $\varepsilon = 1$ in DBSCAN, we achieve exactly the kind of subgraphs we are looking for. We also made use of DBSCAN’s minPts parameter, which controls the minimum number of nodes that have to be contained in a subgraph in order for this subgraph to be considered a hot spot region. We did this because heavy traffic usually spans more than one node, i.e., more than one intersection in the road network. Thus, we used minPts = 3 and minPts = 5 in our experiments. The results of these experiments are listed in Tables 1 and 2.

As is obvious from the figures, the number of predicted hot spot nodes and subgraphs does not increase with increasing number of cars. This demonstrates that hot spots are reliably predicted even for a sample of all moving objects, i.e., based just on the trajectories of 1000 or 2000 cars instead of all 3000. Our results also show that a reasonably small number of subgraphs is derived from a fairly large number of predicted hot spot nodes. The figures in Table 1, i.e., using minPts = 3, show that about 50 subgraphs are detected at any time. These subgraphs represent groups of cars that are driving towards the same destination. The individual groups either approach the destination from different directions, or they are at different distances to the destination. That is, one groups is further away from the destination than another, and thus they form separate hot spots.

	t_{10}		t_{20}		t_{30}		t_{40}	
n	$ hn $	$ hg $	$ hn $	$ hg $	$ hn $	$ hg $	$ hn $	$ hg $
1000	319	5	383	6	526	4	638	5
2000	328	3	478	4	571	3	660	4
3000	358	4	502	4	625	2	683	4

Table 2: Number of predicted hot spot nodes and subgraphs using $\varepsilon = 1$ and $\text{minPts} = 5$

6 Conclusion and Future Work

In this paper we presented an approach for online prediction of hot spots in road networks. As a first step, our OPS approach determines individual hot spot nodes by using an efficient heuristic to predict the traffic intensity at all nodes in the road network. For this, likely visited nodes are predicted for each moving object based on its two most recent positions. Weights are assigned to each of the likely visited nodes such that the sum of all weights for one node represents the expected traffic intensity at that node during a time interval of length h starting at the time of prediction. Based on the hot spots nodes that are predicted by OPS, we then inferred subgraphs that represent hot spot regions within the road network and identified different types of hot spot patterns. In our extensive evaluation using a real large-scale road network, we demonstrated the efficiency and effectiveness of our approach, especially in terms of the number of considered nodes. We also presented an evaluation of identified hot spot nodes and the according derived subgraphs.

As part of our ongoing work, we try to adjust the OPS approach to accommodate objects with different reporting rates. In addition, we experiment with other path prediction approaches to determine if they provide better estimates for the likelihood that a moving object will visit a certain node in the future. Such an adjustment to our approach may further improve the prediction of location, extend, and intensity of hot spots. We also currently work on predicting the life-span of hot spots as well as their evolution from one type to another. This comprehensive prediction of hot spots in road networks may benefit motorists as well as practitioners in application domains like logistics and urban planning.

References

- [AGLW08] Mattias Andersson, Joachim Gudmundsson, Patrick Laube, and Thomas Wollé. Reporting Leaders and Followers among Trajectories of Moving Point Objects. *Geoinformatica*, 12(4), 2008.
- [AT10] Htoo Htet Aung and Kian-Lee Tan. Discovery of Evolving Convoys. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, pages 196–213, 2010.
- [EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

- [FPP97] David Freedman, Robert Pisani, and Roger Purves. *Statistics*. W. W. Norton & Company, 1997.
- [GvK06] Joachim Gudmundsson and Marc van Kreveld. Computing Longest Duration Flocks in Trajectory Data. In *Proceedings of the ACM international symposium on Advances in geographic information systems*, pages 35 – 42, 2006.
- [HLT10] Jiawei Han, Zhenhui Li, and Lu An Tang. Mining Moving Object, Trajectory and Traffic Data (Tutorial at the 15th International Conference Database Systems for Advanced Applications, DASFAA), 2010.
- [HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.
- [JYZ⁺08] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, Christian S. Jensen, and Heng Tao Shen. Discovery of Convoys in Trajectory Databases. *Proceedings of the VLDB Endowment*, 1(1):1068–1080, 2008.
- [JYZJ10] H. Jeung, M. L. Yiu, X. Zhou, and C. S. Jensen. Statistics-Based Path Prediction and Predictive Range Querying in Spatial Network Databases. *The VLDB Journal*, 2010.
- [KMB05] Panos Kalnis, Nikos Mamoulis, and Spiridon Bakiras. On Discovering Moving Clusters in Spatio-temporal Data. In *Proceedings of the International Symposium on Spatial and Temporal Databases*, volume 3633 of *Lecture Notes in Computer Science*, pages 364–381, 2005.
- [KRSZ08] Hans-Peter Kriegel, Matthias Renz, Matthias Schubert, and Andreas Züfle. Statistical Density Prediction in Traffic Networks. In *Proceedings of the SIAM International Conference on Data Mining*, pages 692–703, 2008.
- [KWK⁺07] Sang-Wook Kim, Jung-Im Won, Jong-Dae Kim, Miyoung Shin, Junghoon Lee, and Hanil Kim. Path prediction of moving objects on road networks through analyzing past trajectories. In *KES*, pages 379–389, Berlin, Heidelberg, 2007. Springer-Verlag.
- [LHLG07] Xiaolei Li, Jiawei Han, Jae-Gil Lee, and Hector Gonzalez. Traffic Density-Based Discovery of Hot Routes in Road Networks. In *Proceedings of the International Symposium on Spatial and Temporal Databases*, volume 4605 of *Lecture Notes in Computer Science*, pages 441–459, 2007.
- [LLN⁺10] Siyuan Liu, Yunhuai Liu, Lionel M. Ni, Jianping Fan, and Minglu Li. Towards Mobility-based Clustering. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 919–928, New York, NY, USA, 2010. ACM.
- [OSM] OpenStreetMap. <http://planet.openstreetmap.org>.
- [PvJ06] Mindaugas Pelanis, Simonas Šaltenis, and Christian S. Jensen. Indexing the past, present, and anticipated future positions of moving objects. *ACM Trans. Database Syst.*, 31(1):255–298, 2006.
- [TFPL04] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2004.

Advanced Cardinality Estimation in the XML Query Graph Model

Andreas M. Weiner
Databases and Information Systems Group
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
weiner@cs.uni-kl.de

Abstract: Reliable cardinality estimation is one of the key prerequisites for effective cost-based query optimization in database systems. The XML Query Graph Model (XQGM) is a tuple-based XQuery algebra that can be used to represent XQuery expressions in native XML database management systems. This paper enhances previous works on reliable cardinality estimation for XQuery and introduces several inference rules that deal with the unique features of XQGM, such as native support for Structural Joins, nesting, and multi-way merging. These rules allow to estimate the runtime cardinalities of XQGM operators. Using this approach, we can support classical join re-ordering with appropriate statistical information, perform cost-based query unnesting, and help to find the best evaluation strategy for value-based joins. The effectiveness of our approach for query optimization is evaluated using the query optimizer of XTC.

1 Introduction

Native XML database management systems (XDBMSs) can only become a respected competitor for relational-based XQuery evaluation engines, if they can make the most out of the sophisticated join operators (Structural Joins and Holistic Twig Joins) and indexes (element, content, path, and hybrid indexes) that have been proposed in recent years. A cost-based query optimizer is one of the most important parts of modern database systems that generates alternative query execution plans (QEPs). Such plans can be judged based on their expected execution costs. However, the most challenging task for the optimizer is finding the cheapest plan.

Providing reliable cardinality estimates is key for efficient cost assignment. XQuery is the predominant query language in native XDBMSs. Internally, query languages are represented as logical algebra expressions. The *XML Query Graph Model* (XQGM) [Mat09] is a logical XQuery algebra serving as extension of the seminal Query Graph Model [PHH92] and introduces an extended XQuery data model, supports query de-correlation, and provides seamless support for Structural Joins (SJs) [AKJP⁺02] and Holistic Twig Joins (HTJs) [BKS02].

Most surprisingly, almost all cardinality estimation frameworks for native XDBMSs only deal with the estimation of XPath path cardinalities. Even though XPath covers an important fragment of XQuery, restricting cardinality estimation to it is insufficient. To the best

of our knowledge, the work of Teubner et al. [TGMS08] is the only one that tackles this important problem in the context of relational XQuery engines. Even though many parts of their approach can be transferred to native XDBMSs, further adjustments are necessary: Amongst others, XQGM natively supports Full, Semi, and Outer SJs as logical algebra building blocks.

1.1 Related Work

Cost-based query optimization in semi-structured database systems emerged in the context of the Lore project [MAG⁺97]. In Lore, DataGuides [GW97] provide a simple means for managing the cardinalities of unique paths in XML documents. XTC's statistics manager reuses this principle to provide the basic statistical information needed to bootstrap our cardinality inference rules. If there are non-unique paths in an XML document, the DataGuide is insufficient to provide the necessary information. In the course of time, many researchers proposed concepts for estimating the path cardinalities in such situations, e. g., [AAN01, FHR⁺02, ZÖAI06, BEH⁺06, FM07, AH08]. These approaches are mostly focusing on estimation accuracy and on minimal space consumption of their data structures. Even though path expressions are important building blocks of XQuery, these approaches do not help to optimize more complex queries (see Section 2.2).

The early work of Sartiani [Sar03] claims to discuss cardinality estimation of FLWR expressions, but focuses mostly on `for` expressions.

Having a look at native XML database management systems that provide cost-based query optimizers, e. g., Natix [FHK⁺02] or Timber [JAKC⁺02], shows that their cardinality estimation capabilities are restricted to cardinality estimation of simple path expressions. To the best of our knowledge, MonetDB/XQuery, and its respective XQuery compiler *Pathfinder*, is the only (non-native) XML database management system that supports XQuery cardinality estimation [TGMS08]. They use the general approach of abstract domain identifiers to estimate the value space that is taken by tuple items at runtime. As their cardinality inference rules are strongly tied to their logical algebra, it cannot be directly used in the context of XQGM. In contrast, our work reuses their concept of abstract domain identifiers, but introduces a novel set of inference rules that allow to gain reliable cardinality estimates for XQGM instances.

1.2 Contribution

The contribution of this paper can be summarized as follows:

- We will derive a set of inference rules that allow to perform XQuery cardinality estimation on XQGM instances.
- We will discuss the optimization of value-based joins, which especially require reliable estimates to choose the right join operator and join order to prevent bad plans.
- We describe how reliable cardinality estimates allow for cost-based query unnesting in XTC.

- We show that our inference rules provide reliable cardinality estimates for a wide range of queries and support our cost-based query optimizer in providing linear scalability for the XMark benchmark queries.

2 Preliminaries

Before we detail our inference rules, Section 2.1 briefly introduces XQGM, which is our internal representation for XQuery expressions. Thereafter, Section 2.2 motivates the importance of reliable cardinality estimation for efficient query evaluation in native XDBMSs.

2.1 A Brief Introduction to XQGM

XQGM is the logical XQuery algebra of the *XML Transaction Coordinator* (XTC) [HMB⁺10], which is our prototype of a native XDBMS supporting, amongst others, ACID transactions and cost-based query optimization [WHdS10].

2.1.1 XQGM Data Model

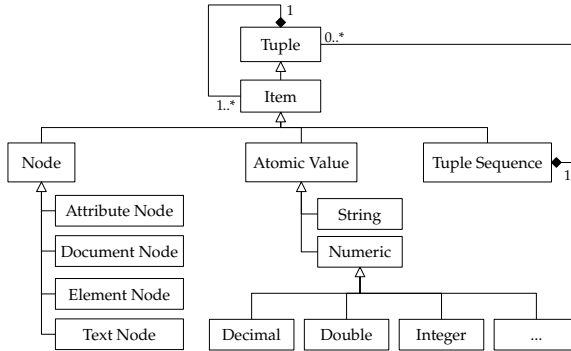


Figure 1: The XQGM data model [Mat09]

The XQGM data model extends the *XQuery Data Model* (XDM) [FMM⁺07], which supports *atomic values*, *items*, and *sequences* as model primitives. The XQGM data model supplements these primitives by *tuples*. More precisely, every XQGM data model object is a tuple [Mat09]. Figure 1 depicts the UML diagram of the XQGM data model components. An item can be (1) a node, (2) an atomic value, or (3) an ordered list of tuples, which we denote as *tuple sequence*. Thus, tuple sequences can contain nested tuples. We refer to a tuple sequence that contains exactly one tuple as a *singleton tuple sequence* and a tuple

that is formed by a single item is called a *singleton tuple*¹. A singleton is always equal to the unique element it contains. Moreover, a tuple sequence is an XQuery sequence, if and only if it contains only singleton tuples or is equal to an empty sequence [Mat09]. During query evaluation, we postulate that only the final result must be an XQuery sequence, whereas intermediate results must not conform to the XDM, but only to the XQGM data model.

2.1.2 XQuery the XQGM Way

Before an XQGM instance is passed on to our cost-based query optimizer [WHdS10], several algebraic rewrites are applied. One of the major challenges for native XML query processing is bridging the gap between two completely different processing strategies: *node-at-a-time* and *set-at-a-time* processing. Node-at-a-time evaluation is inherent to the *XQuery Core Language* and follows a nested-loops-style evaluation approach (nested `for` loops) that is similar to sub-selects in SQL [Mat07]. Even though it is not very efficient in most cases, it can be beneficial in low-selectivity scenarios. On the other hand, set-at-a-time query evaluation is employed by almost all SJ and HTJ algorithms and is in most cases very efficient.

XQGM supports both processing strategies. After an initial translation of an XQuery expression into XQGM, query unnesting tries to iteratively replace path expressions by cascades of (logical) SJs. During this rewriting process, further operators may be introduced, e. g., outer join operators helping to preserve the correct output semantics for positional predicates or for the `return` clause in the presence of empty sequences.

2.1.3 XQGM Example

Figure 2 illustrates an XQGM instance for the following query, which returns the `author` names of `book` nodes whose `price` is larger than 1.99:

```
<result> {
  for $b in doc("sample.xml")//book
  where $b/price > "1.99"
  return
    <author>{ $b/author/text() }</author>
} </result>
```

In XQGM, there exists no direct connection between operators. Instead, every operator contains so-called *tuple variables* that actually receive the tuple sequences emitted by child operators. We distinguish between three different types of quantifiers: *for* (F), *let* (L), and *exists* (E). The first and the second quantifier have a similar semantics as the corresponding XQuery constructs: A *for*-quantified tuple variable iterates over the tuple sequence it receives from its input. In contrast, *let*-quantified tuple variables deliver their connected sequences at once. Finally, *exists*-quantified tuple variables are used to check whether their inputs are non-empty tuple sequences.

¹For the rest of this work, whenever the context is unambiguous, we just refer to it as singleton.

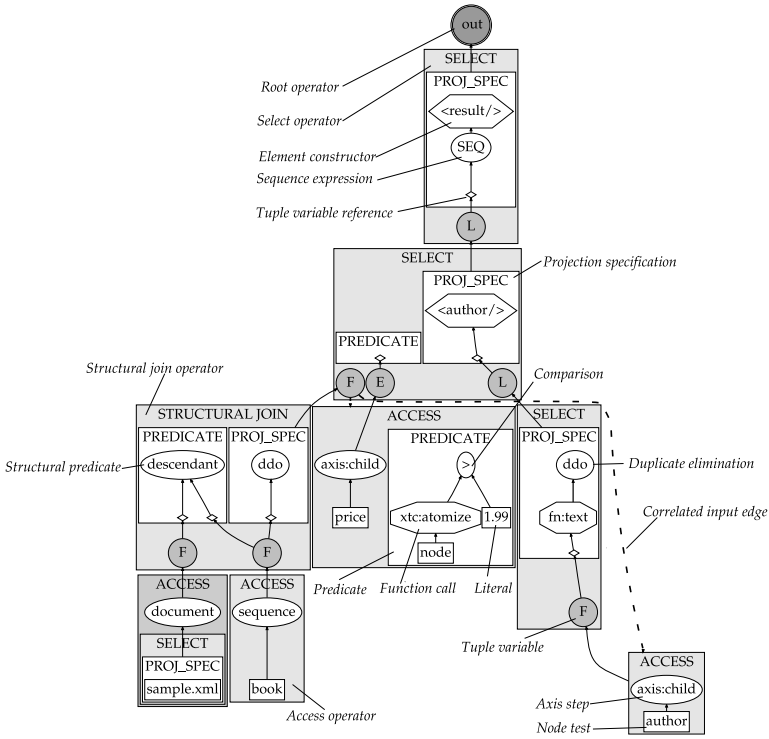


Figure 2: Sample XQGM instance

Query evaluation starts at the left-most subtree (Document Access operator). The SJ operator evaluates the structural predicate (*descendant* axis) between the virtual document root and all *book* elements. Actually, SJ is a Semi Join operator, because only the tuple variable that provides *book* elements is linked to the projection specification (PROJ_SPEC). The parent Select operator receives a tuple sequence of all *book* elements and iterates over it: First, it sends the current evaluation context to the Access operator associated with it via the *exists*-quantified tuple variable². The Access operator selects a sequence of all *price* element nodes that are children of the current (context) *book* element and returns only those items that satisfy the predicate. Next, the Select operator passes the current context to the right-most Access operator that returns all *author* nodes that are connected to the current context node via the *child* axis. For each match, the parent Select operator evaluates the *fn:text()* function. Now, the Select operator, which is at the heart of the query graph, can produce an output: It creates a new XML element `<authors>seq</authors>`, if the sequence bound to the *exists*-quantified tuple variable is non-empty, whereas *seq* represents the tuple sequence received by the

²In the graphical XQGM representation, the context passing is illustrated as a dashed line from the “sending” tuple variable to the receiving operator.

let-quantified tuple variable. Finally, the top-most Select operator wraps its input in an opening `<result>` and a closing `</result>` tag.

2.2 Problem Statement

Today, cardinality estimation frameworks for native XDBMSs are not prepared to handle XQuery expressions sufficiently. Most of them are only focusing on estimating the output sizes of simple XPath expressions. Let us consider a slightly simplified query from the W3C XQuery Uses Cases query set (Use Case “R”, query Q 13):

```
<result> {
  for $uid in distinct-values(doc("bids.xml")//userid),
  $u in doc("users.xml")//user_tuple[userid = $uid]
  let $b := doc("bids.xml")//bid_tuple[userid = $uid]
  return
    <bidder name="{ $u/name}" bidCount="{ count($b) }" />
} </result>
```

The query returns for each user having placed a bid the corresponding user name and the total number of bids. Figure 3 shows the graphical representation of the respective XQGM instance.

The subtree rooted at operator **1** delivers the result for the expression `distinct-values(doc("bids.xml")//userid)`. The Select operator **2** provides the “heart-beat” for the further workflow. Operator **3** evaluates the first value-based join. Operator **4** triggers the evaluation of the second expression in the `for` clause that is bound to `$u`. The Structural Outer Join (operator **5**) returns `(userid, user_tuple)` tuples and preserves `user_tuple` nodes even if it does not find a matching join partner (necessary for preserving the correct output semantics for empty sequences in the final `return` clause). After evaluating the value-based join, the qualifying `user_tuple` nodes are passed on to operator **6** that provides the results for the `name` attribute in the query result (operator **7**).

In the `let` clause of our sample query, there is a second value-based join (operator **10**). Operator **8** furnishes the evaluation context (dashed line) for the Access operator below operator **9**. Finally, the result is passed from operator **7** to operator **2**, which, in turn, sends it to operator **11**.

Even for this simple XQuery expression, present cardinality estimation frameworks fail to provide satisfying results. These frameworks are only capable of providing estimates for the outputs of the XPath expressions bound to `$uid` (see Section 1.1). This situation is really sobering, because many optimization decisions, e.g., the selection of appropriate join operators for value-based joins or Structural Join reordering can, in such cases, only be based on vague and coarse-grained heuristics. For example, if a value-based join could be evaluated using a hash join operator, we could only guess which input should be hashed and which one probed. Here, a wrong guess could lead to tremendous performance loss. Moreover, we cannot provide cardinality estimates for the final query result. For example, this might have an impact on the selection of an appropriate materialization

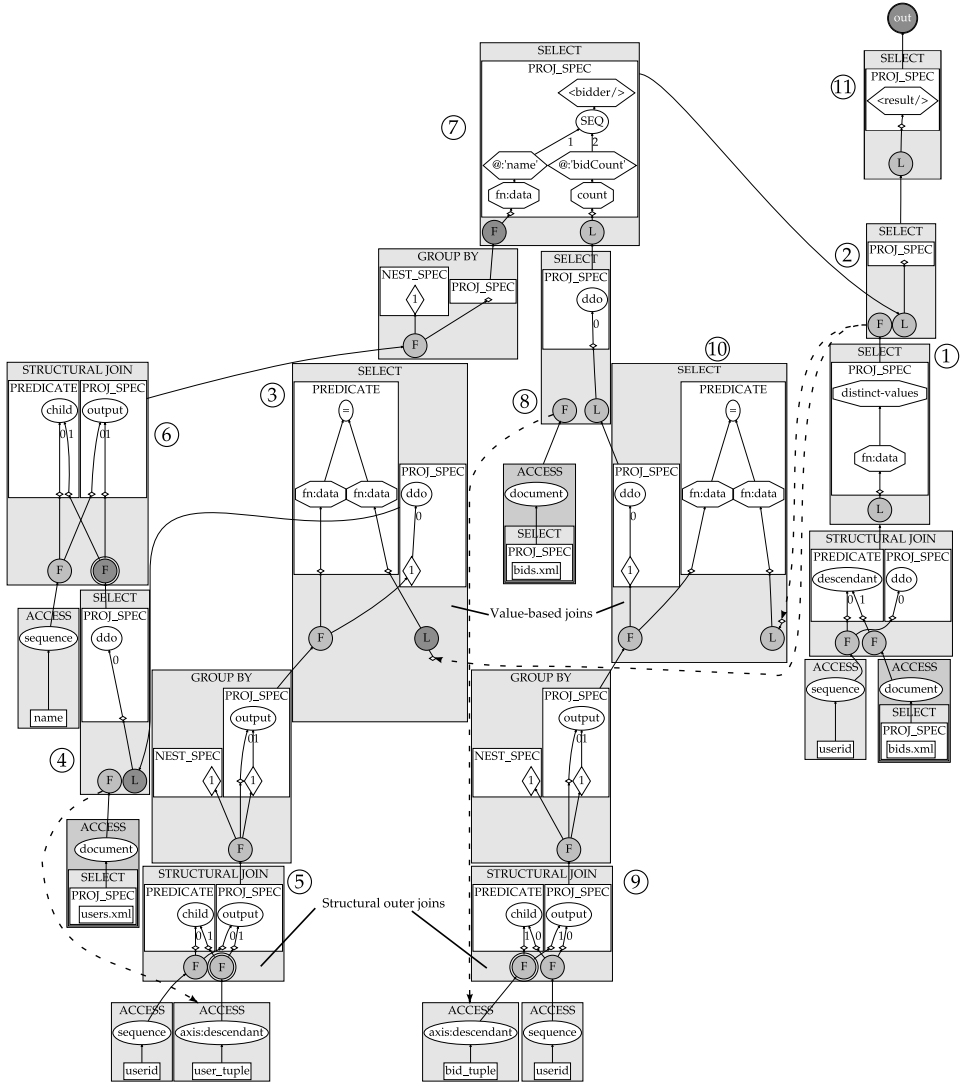


Figure 3: XQGM instance for the sample query

strategy: If there are only few final results, late materialization might be preferred over early materialization.

Even though the concept of *abstract domain identifiers* and the corresponding cardinality inference rules introduced by Teubner et al. [TGMS08] provide an elegant means for gaining reliable cardinality estimates, they rely on a completely different algebra and, hence, cannot be directly used out-of-the-box for our purposes. Consequently, a similar set of inference rules must be specified that provide support for XQGM-specific operators such as logical Structural Joins, n -way outer joins, and unnest operators to allow for full cardinality estimation on XQGM instances.

At the moment, XTC’s query optimizer applies query unnesting [Mat07] as a heuristics. If we could provide reliable cardinality estimates for XQGM, we would be able to leave the decision whether to perform query unnesting or not to the cost-based query optimizer, which would make the query optimization process even more flexible.

3 Application Scenarios for the Inference Rules

In this section, we have a brief look at two applications of our cardinality inference rules that complement the classical scenario for cardinality estimation, i. e., providing reliable statistical information for join reordering.

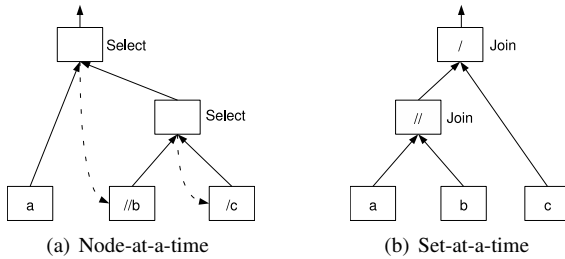


Figure 4: Evaluation strategies for XML queries

In Section 2.1.2, we mentioned the importance of query unnesting for efficient query evaluation. Figure 4(a) shows how a simple XPath expression $a//b/c$ is evaluated using node-at-a-time and set-at-a-time processing, respectively. During node-at-a-time processing, for every a node, the evaluation context for the evaluation of $//b$ is provided (dashed line). By iterating over all qualified b nodes, the evaluation context for $/c$ is furnished. Finally, every qualified c node is output. On the other hand, set-at-a-time query evaluation works similar to relational merge joins. Figure 4(b) illustrates how the XPath expression $a//b/c$ is evaluated using this approach: First, the location step $//$ is evaluated between all a and b nodes. Afterwards, the result of the first Structural Join operator serves together with all c nodes as input for the second one that evaluates the location step b/c . In XTC, the transition from node-at-a-time to set-at-a-time processing is performed during

a pattern-based algebraic rewrite process³. So far, query rewrites are applied in an eager way, i.e., as long as query unnesting is possible, it is applied. Now, using our refined cardinality information, the query optimizer can immediately abort query unnesting if the estimated cardinalities indicate that this would be counterproductive.

Efficient evaluation of value-based joins is crucial for the performance of XML database systems. In contrast to structural predicates, which can be decided using appropriate node labeling schemes without any further access to the document, value-based predicates need additional accesses to the document to fetch the actual content nodes. Let us have a look at the XQGM instance shown in Figure 3. For example, operator **3** evaluates a value-based predicate. At the logical level, for each tuple bound to the *let*-quantified tuple variable, a complete iteration over the sub-expression rooted at the *for*-quantified tuple variable is performed. Choosing the right physical implementation is crucial for the query performance. During optimization, it is especially important to know how many tuples will be delivered via the *for*-quantified and *let*-quantified tuple variables. Hence, the evaluation order proposed by the logical algebra is not always the most efficient one: Let us assume that the optimizer can use a value-based hash join for evaluating the value-based join. If we would know that the *let*-quantified tuple variable would deliver 1,000 tuples, but the *for*-quantified tuple variable only 10, then it would be much cheaper to hash the 10 tuples (only a single evaluation of the sub-expression) and probe the input of the *let*-quantified tuple variable against it. But for a reciprocal input ratio, this decision would result in an extremely slow QEP.

4 Cardinality Inference

In this section, we introduce the various inference rules that are used for cardinality estimation in XTC. Section 4.1 introduces preliminary notions. Thereafter, Section 4.2 discusses the inference rules.

4.1 Nomenclature

Our cardinality estimation approach is based on the idea of *abstract domain identifiers* [TGMS08]. For your convenience, we repeat its definition and adjust it to our needs in the context of XQGM. Abstract domain identifiers, which will be denoted by Greek letters such as α, β, \dots , allow to estimate the value space of tuple items that are exposed by XQGM expressions at runtime.

Each XQGM operator consumes or emits tuple sequences. Let $s = \langle t_1, \dots, t_n \rangle$ be a tuple sequence where each $t_j = [i_{j1}, \dots, i_{jm}]$ is a tuple with m items. Obviously, this tuple sequence has a “fixed schema” with m columns (denoted by c_1, \dots, c_m), i.e., every tuple has m (probably nested) items where each i_{jk} shares a common domain (for an arbitrarily but consistently chosen k). Table 1 illustrates this hypothetical schema.

Let us denote the *active domain* of c_i , i.e., the set of all values taken by tuples $t_1 \dots, t_n$ in

³The example shown here is very simplistic and abstract. In reality, the patterns are much more complex and may cover numerous operators. However, the rationale behind the two evaluation strategies should be obvious.

	c_1	\dots	c_m
t_1	i_{11}	\dots	i_{1m}
\vdots	\vdots	\ddots	\vdots
t_n	i_{n1}	\dots	i_{nm}

Table 1: Hypothetical schema

column c_i , by α_i . Moreover, we refer to the *domain size*, i.e., the total number of distinct values in c_i , by $\|\alpha_i\|$. We define $\text{dom}(o) = \{c_1^{\alpha_1}, \dots, c_m^{\alpha_m}\}$ as the *result domain set* of the tuple sequence produced by XQGM operator o as output⁴, whereas $c_i^{\alpha_i}$ is the i -th column of the tuple sequence emitted by o with the corresponding active domain α_i .

According to [TGMS08], for the abstract domain identifiers α and β , we define the reflexive and transitive *inclusion relationship* $\beta \sqsubseteq \alpha$ as follows: $\beta \sqsubseteq \alpha \iff \forall b \in \beta : b \in \alpha$. Finally, we will denote the assignment of an inferred value by $=^!$ and an inferred inclusion relationship by $\sqsubseteq^!$.

4.2 Inference Rules

After providing the preliminaries, we are now ready to discuss the different inference rules that are used to estimate the cardinalities of XQGM (sub)-expressions. The definition of the various inference rules is mostly based on the notation introduced by [TGMS08].

4.2.1 Access Operators

Figure 5 shows the inference rules for Access operators. In XQGM, query evaluation starts at a document's root node. Rule CARD-DOC-ACCESS simply assigns the value 1 to the abstract domain identifier α , because there is only a single document root in the document.

Rule CARD-ACCESS is responsible for deriving the cardinality estimates for basic Access operators. An Access operator provides a tuple stream having an element or attribute name e as filter and may evaluate an optional predicate p . We estimate its cardinality by the total number of element or attribute names having the corresponding name. The selectivity of predicate p can be determined by employing histograms or by simply using the famous 10 % heuristics of System R [SAC⁺79].

Finally, CARD-ACCESS-WITH-CONTEXT serves for estimating the cardinality of access operators whose output depends on an evaluation context. Such operators are employed for XQGM sub-expressions that cannot be unnested according to the unnesting rules defined by Mathis [Mat07]. Here, the abstract domain identifier β of the context-providing operator is used as a starting point for cardinality inference. In expression $\sigma(b \theta e)$, b is

⁴Please note, here we assume that the identifiers $\alpha_1, \dots, \alpha_m$ have not been used before.

$$\frac{}{\text{dom}(\text{DocAccess}) = \{c^\alpha \wedge \|\alpha\| = 1\}} \quad (\text{CARD-DOC-ACCESS})$$

$$\frac{}{\text{dom}(\text{Access}_e p) = \{c^\alpha \wedge \|\alpha\| = 1 \mid e \mid \cdot \sigma(p)\}} \quad (\text{CARD-ACCESS})$$

$$\frac{b^\beta \in \text{dom}(\square) \wedge \square \text{ provides evaluation context}}{\text{dom}(\text{Access}_{e\theta}) = \{c^\alpha \wedge \|\alpha\| = 1 \mid e \mid \cdot \sigma(b\theta e)\}} \quad (\text{CARD-ACCESS-WITH-CONTEXT})$$

Figure 5: Inference rules for access operators

the current context item, θ is the corresponding XPath axis and e is the tuple stream issued by the current access operator.

For example, in Figure 3, the left-most Access operator (below sub-expression 1) accesses the sequence of all `userid` nodes and does not evaluate a predicate. The right-most operator simply accesses the document root of document `bids.xml`. Sub-expression 9 in Figure 3 shows an Access operator (left-most access operator) that is a match for rule CARD-ACCESS-WITH-CONTEXT. Here, The Select operator (Operator 8) provides the evaluation context, θ is the XPath `descendant` axis, and e corresponds to the element name `bidtuple`.

4.2.2 Cardinality Estimation for Structural Joins

In XQGM, Structural Joins are the basic building blocks for the evaluation of path expressions. Whenever possible, we use Structural Semi Joins to reduce intermediate results. Figure 6 illustrates the inference rules for the six different Structural Join types. For the representation of the various join types, we use the well-known symbols: \bowtie_p (Structural Left Semi Join), \bowtie_p (Structural Right Semi Join), \bowtie_p (Structural Full Join), \bowtie_p (Structural Left-outer Join), and \bowtie_p (Structural Right-outer Join). The Structural Join evaluates a structural predicate p described as follows: $a_i \theta b_j$, where a_i is an item of tuple sequence q_1 , b_j is an item of tuple sequence q_2 , and θ is an XPath axis, e. g., `descendant`.

Even though the definitions of the inference rules seem to be cumbersome at first sight, their rationale is very simple: For estimating the result size of active domains affected by the structural predicate, we rely on two data structures. In the case of path expressions describing linear and unique paths, we use the *path synopsis*, which is an extension of the seminal DataGuide [GW97], to derive accurate cardinalities for the expression. If the path expression is more complex or involves non-unique paths, we approximate the cardinality using XTC's XPath cardinality estimation framework called *EXsum* [AH08].

The expression $\sigma(a_i[\theta b_j])$ returns the selectivity of a_i items connected to b_j items via the θ axis, i. e., the percentage of a_i nodes satisfying the structural predicate. On the other hand, $\sigma(a_i \theta b_j)$ returns the selectivity of b_j items. For the remaining items, we follow the idea of Teubner et al. [TGMS08] to use a generalization of the classical 10% rule to estimate the new cardinalities of active domains that are not directly affected (i. e.,

$$\frac{a_i^{\alpha_i} \in \text{dom}(q_1) \wedge b_j^{\beta_j} \in \text{dom}(q_2)}{|q_1 \bowtie_{a_i \theta b_j} q_2| = \|\alpha_i\| \cdot \sigma(a_i[\theta b_j]) \wedge \text{dom}(q_1 \bowtie_{a_i \theta b_j} q_2) = \left\{ c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \text{dom}(q_1) \setminus \{a_i^{\alpha_i}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1 - (1 - 1/10)^{|q_1|/\|\gamma_1\|}] \right\} \cup \left\{ c^\gamma \mid \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! |q_1 \bowtie_{a_i \theta b_j} q_2| \right\}} \quad (\text{CARD-SJ-1})$$

$$\frac{a_i^{\alpha_i} \in \text{dom}(q_1) \wedge b_j^{\beta_j} \in \text{dom}(q_2)}{|q_1 \bowtie_{a_i \theta b_j} q_2| = \|\beta_j\| \cdot \sigma(a_i[\theta b_j]) \wedge \text{dom}(q_1 \bowtie_{a_i \theta b_j} q_2) = \left\{ c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \text{dom}(q_2) \setminus \{b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1 - (1 - 1/10)^{|q_2|/\|\gamma_1\|}] \right\} \cup \left\{ c^\gamma \mid \gamma \sqsubseteq^! \beta_j \wedge \|\gamma\| =^! |q_1 \bowtie_{a_i \theta b_j} q_2| \right\}} \quad (\text{CARD-SJ-2})$$

$$\frac{a_i^{\alpha_i} \in \text{dom}(q_1) \wedge b_j^{\beta_j} \in \text{dom}(q_2) \wedge a_i \theta b_j \text{ is location step}}{|q_1 \bowtie_{a_i \theta b_j} q_2| = \|\beta_j\| \cdot \sigma(a_i[\theta b_j]) \wedge \text{dom}(q_1 \bowtie_{a_i \theta b_j} q_2) = \left\{ c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \text{dom}(q_1) \cup \text{dom}(q_2) \setminus \{a_i^{\alpha_i}, b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}] \right\} \cup \left\{ c^\gamma \mid \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! \|\alpha_i\| \cdot \sigma(a_i[\theta b_j]) \right\} \cup \left\{ c^\gamma \mid \gamma \sqsubseteq^! \beta_j \wedge \|\gamma\| =^! \|\beta_j\| \cdot \sigma(a_i \theta b_j) \right\}} \quad (\text{CARD-SJ-3})$$

$$\frac{a_i^{\alpha_i} \in \text{dom}(q_1) \wedge b_j^{\beta_j} \in \text{dom}(q_2) \wedge a_i \theta b_j \text{ is predicate step}}{|q_1 \bowtie_{a_i \theta b_j} q_2| = \|\alpha_i\| \cdot \sigma(a_i[\theta b_j]) \wedge \text{dom}(q_1 \bowtie_{a_i \theta b_j} q_2) = \left\{ c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \text{dom}(q_1) \cup \text{dom}(q_2) \setminus \{a_i^{\alpha_i}, b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}] \right\} \cup \left\{ c^\gamma \mid \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! \|\alpha_i\| \cdot \sigma(a_i[\theta b_j]) \right\} \cup \left\{ c^\gamma \mid \gamma \sqsubseteq^! \beta_j \wedge \|\gamma\| =^! \|\beta_j\| \cdot \sigma(a_i \theta b_j) \right\}} \quad (\text{CARD-SJ-4})$$

$$\frac{a_i^{\alpha_i} \in \text{dom}(q_1) \wedge b_j^{\beta_j} \in \text{dom}(q_2)}{|q_1 \bowtie_{a_i \theta b_j} q_2| = \|\beta_j\| \wedge \text{dom}(q_1 \bowtie_{a_i \theta b_j} q_2) = \left\{ b_j^{\beta_j} \right\} \cup \left\{ c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \text{dom}(q_1) \cup \text{dom}(q_2) \setminus \{a_i^{\alpha_i}, b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}] \right\} \cup \left\{ c^\gamma \mid \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! \|\alpha_i\| \cdot \sigma(a_i[\theta b_j]) \right\}} \quad (\text{CARD-SJ-5})$$

$$\frac{a_i^{\alpha_i} \in \text{dom}(q_1) \wedge b_j^{\beta_j} \in \text{dom}(q_2)}{|q_1 \bowtie_{a_i \theta b_j} q_2| = \|\alpha_i\| \wedge \text{dom}(q_1 \bowtie_{a_i \theta b_j} q_2) = \left\{ a_i^{\alpha_i} \right\} \cup \left\{ c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \text{dom}(q_1) \cup \text{dom}(q_2) \setminus \{a_i^{\alpha_i}, b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}] \right\} \cup \left\{ c^\gamma \mid \gamma \sqsubseteq^! \beta_j \wedge \|\gamma\| =^! \|\beta_j\| \cdot \sigma(a_i \theta b_j) \right\}} \quad (\text{CARD-SJ-6})$$

Figure 6: Inference rules for Structural Joins

independent) by the structural predicate: $\|\gamma_2\| =^! \|\gamma_1\| \cdot [1 - (1 - 1/10)^{|q|/\|\gamma_1\|}]$, where $|q|$ is the cardinality of input operator q , γ_1 is the active domain cardinality of a column contained in the input tuple sequence of q , and $\|\gamma_2\|$ is the inferred cardinality for the corresponding active domain in the output tuple sequence.

Rules CARD-SJ-1 and CARD-SJ-2 depict the inference rules for Structural Left Semi Joins and Right Semi Joins, respectively. Here, the result domain set is equal to the domain set of the left or right input operator, respectively. The cardinalities of the active domains' join

items are calculated using the path synopsis or EXsum and all remaining cardinalities of the active domains in the output domain set are approximated using the generalized 10 % rule.

For the cardinality estimation of Structural Full Joins, rules CARD-SJ-3 and CARD-SJ-4 show the corresponding definitions, which distinguish between the evaluation of location steps and predicate steps. Both rules estimate the output cardinality of the join operator and the active domains of join items a_i and b_j using the path synopsis. Once again, the generalized 10 % rule helps to approximate the active domains for items that are independent of the join predicate.

In Figure 3, several Structural Outer Joins are used (e. g., operators **5**, **6**, and **9**). Inference rules CARD-SJ-5 and CARD-SJ-6 allow for cardinality inference of Structural Left-outer and Structural Right-outer Joins, respectively⁵. The active domain of the join item whose tuple sequence contributes to the output's outer part remains unchanged, and the cardinality of the other join item is adjusted according to EXsum's estimation. For all remaining active domains, the generalized 10 % rule is applied.

4.2.3 Inference Rules for Grouping, Unnesting, and Miscellaneous Operators

$$\begin{array}{ll}
\frac{a_i^{\alpha_i} \in \text{dom}(q)}{|\text{GroupBy}_i(q)| = \|\alpha_i\|} & (\text{CARD-GROUP-BY}) \\
\frac{a_i^{\alpha_i} \in \text{dom}(q)}{|\text{Unnest}_i(q)| = \text{flatCard}(q)} & (\text{CARD-UNNEST}) \\
\frac{\square \in \{\text{Split}, \text{Project}\} \wedge \text{dom}(q') \subseteq \text{dom}(q)}{\text{dom}(\square(q)) = \{a_i^{\alpha_i} \mid a_i^{\alpha_i} \in \text{dom}(q')\}} & (\text{CARD-MISC-1}) \\
\frac{\square \in \{\text{Sort}, \text{DDO}\}}{|\square(q)| = |q| \cdot 2/3} & (\text{CARD-MISC-2})
\end{array}$$

Figure 7: Inference rules for grouping, unnesting, and miscellaneous operators

The GroupBy and the Unnest operator allow for nesting and unnesting of tuple sequences w. r. t. a specific item position, respectively. Figure 7 (CARD-GROUP-BY) shows the cardinality inference for nesting. The cardinality of the result tuple sequence is primarily determined by the cardinality of the active domain according to which the nesting is performed. The active domain set remains unchanged.

The cardinality inference rule for the Unnest operator (Figure 7 CARD-UNNEST) approximates the cardinality of the result tuple sequence using the function *flatCard*, which calculates the cardinality of the Cartesian product of the values of item i and the values of the remaining active domains.

The Split operator sends its input to multiple consumers and the Project operator serves as classical projection operator. Figure 7 (CARD-MISC-1) simply derives the result domain set by considering those active domains that are referred to in the projection specification, where q is the input tuple sequence and q' is the output tuple sequence. Again, the cardinality remains unchanged.

⁵Please note, we do not use Structural Full-outer Joins in XQGM.

Conventionally, every XQGM operator returns tuple sequences sorted in document order and tries to reduce duplicates to a minimum. If duplicates cannot be avoided, e.g., if a full join using the `descendant` axis is performed, additional duplicate elimination might become necessary. Sort and Distinct-Doc-Order (DDO) retain a certain sort order or eliminate duplicates. Inference rule **CARD-MISC-2**, depicted in Figure 7, describes the estimation of the output cardinality. We expect that most tuple sequences are almost duplicate free and sorted, therefore, we assume that two thirds of their input tuples will “survive”.

4.2.4 Inference Rules for Merge and Select

Rules **CARD-MERGE-1** and **CARD-MERGE-2**, listed in Figure 8, show the inference rules for the Merge operator. The Merge operator contains only *for*-quantified tuple variables and calculates the Cartesian product on its input tuple streams. Each Merge operator contains a so-called merge specification that describes a complex selection predicate on the Cartesian product. The predicate selects all tuples that have equal values for given positions in the tuple sequence. For the sake of simplicity, we also use the 10 % rule to determine the output cardinality.

$$\frac{}{|\text{Merge}(q_1, \dots, q_n)| = \prod_{i=1}^n |q_i| \cdot 1/10 \wedge \text{dom}(\text{Merge}(q_1, \dots, q_n)) \supseteq \{c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \cup_{j=1}^n \text{dom}(q_j) \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1 - (1 - 1/10)^{\sum_{i=1}^n |q_i| / \|\gamma_1\|}]\}} \quad (\text{CARD-MERGE-1})$$

$$\frac{q_i \text{ delivers outer sequence}}{|\text{Merge}(q_1, \dots, q_n)| = |q_i| \wedge \text{dom}(\text{Merge}(q_1, \dots, q_n)) \supseteq \{c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \cup_{j=1}^n \text{dom}(q_j) \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1 - (1 - 1/10)^{|q_i| / \|\gamma_1\|}]\}} \quad (\text{CARD-MERGE-2})$$

$$\frac{}{|\text{Select}_p(q_1, \dots, q_n)| = \prod_{q \in Q_{\text{for}} \cup Q_{\text{let}}} |q| \cdot 1/10 \wedge \text{dom}(\text{Select}(q_1, \dots, q_n)) = Q'_{\text{for}} \cup Q'_{\text{let}} \cup Q'_{\text{exists}} \wedge Q'_{\text{for}} = \{c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq \gamma_j \wedge c_1^{\gamma_1} \in \text{dom}(q_j) \wedge q_j \in Q_{\text{for}} \wedge \|\gamma_2\| =^! \|\gamma_j\| \cdot [1 - (1 - 1/10)^{|q_j| / \|\gamma_j\|}]\} \wedge Q'_{\text{let}} = \{c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq \gamma_j \wedge c_1^{\gamma_1} \in \text{dom}(q_j) \wedge q_j \in Q_{\text{let}} \wedge \|\gamma_2\| =^! \|\gamma_j\| \cdot [1 - (1 - 1/10)^{|q_j| / \|\gamma_j\|}]\} \wedge Q'_{\text{exists}} = \{c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq \gamma_j \wedge c_1^{\gamma_1} \in \text{dom}(q_j) \wedge q_j \in Q_{\text{exists}} \wedge \|\gamma_2\| =^! \|\gamma_j\| \cdot [1 - (1 - 1/2)^{|q_j| / \|\gamma_j\|}]\}} \quad (\text{CARD-SEL})$$

Figure 8: Inference rules for merge and select

Rule **CARD-MERGE-2** handles a special case: If there exists an outer tuple variable in the Merge operator, the outer semantics well-known from outer joins is used, i.e., for every tuple sequence where a match does not exist, the tuple still appears in the Cartesian product and all non-matching items are replaced by empty sequences [Mat09]. Here, the output cardinality is simply determined by the cardinality of the tuple sequence associated with the operator connected to the outer tuple variable.

The Select operator also calculates the Cartesian product on its input streams. In contrast to the Merge operator, the Select operator can contain tuple variables with mixed quantifiers: *for*, *let*, or *exists*. The Select operator is the most versatile operator in XQGM, as it allows

to express value-based joins and simple selection predicates as well as triggering XQuery *for-let* bindings. Rule CARD-SEL describes the cardinality inference for Select operators, where Q_q (Q'_q) is the set of all columns that are connected to q -quantified (output) tuple variables. The *for*-quantified tuple variables “drive” the output generation process. On the other hand, the tuple sequences bound to *let*-quantified tuple variables are “nested” into the results generated by *for*-quantified tuple variables, whereas *exists*-quantified tuple variables only serve for existence tests and do not contribute to the output.

4.2.5 Inference Rules for Set Operators

Finally, as shown in Figure 9, XQGM provides three set operators: Union, Intersect, and Difference. In XQGM, Union and Intersect are n -way operators and only Difference is a binary operator.

$$\begin{array}{c}
\frac{|q_1| = |q_2| = \dots = |q_n|}{|\cup_{i=1}^n q_i| = \sum_{i=1}^n |q_i| \wedge \text{dom}(\cup_{i=1}^n q_i) = \cup_{k=1}^{|\text{dom}(q_1)|} \left\{ c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^1 \gamma_1 \wedge c_{1k}^{\gamma_1} \in \text{dom}(q_1) \wedge \|\gamma_2\| =^1 \sum_{i=1}^n \|\gamma_{ik}\| \right\}} \quad (\text{CARD-UNION}) \\
\\
\frac{a_1^{\alpha_1} \in \text{dom}(q_1) \wedge \dots \wedge a_n^{\alpha_n} \in \text{dom}(q_n) \wedge |q_k| = \min\{|q_1|, \dots, |q_n|\}}{|\cap_{i=1}^n q_i| = |q_k| \cdot 2/3 \wedge \text{dom}(\cap_{i=1}^n q_i) = \left\{ c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^1 \gamma_1 \wedge c_1^{\gamma_1} \in \text{dom}(q_k) \wedge \|\gamma_2\| =^1 \|\gamma_1\| \cdot [1 - (1 - 1/10)^{q_k / \|\gamma_1\|}] \right\}} \quad (\text{CARD-INTERSECT}) \\
\\
\frac{|q_1 \setminus q_2| = |q_1| \cdot 1/10 \wedge \text{dom}(q_1 \setminus q_2) = \left\{ c_2^{\gamma_2} \mid \gamma_2 \sqsubseteq^1 \gamma_1 \wedge c_1^{\gamma_1} \in \text{dom}(q_1) \wedge c_1^{\gamma_1} \notin \text{dom}(q_2) \wedge \|\gamma_2\| =^1 \|\gamma_1\| \cdot [1 - (1 - 1/10)^{q_1 / \|\gamma_1\|}] \right\}}{ \quad } \quad (\text{CARD-DIFFERENCE})
\end{array}$$

Figure 9: Inference rules for set operators

Rule CARD-UNION describes the cardinality inference for the Union operator. Here, we assume that all input operators ($q_1 \dots q_n$) have the same output cardinality and all input tuple sequences have the same active domains whose value ranges may differ. In this case, γ_{ik} denotes the active domain of operator i in column k .

The cardinality inference for the Intersect operator is described by rule CARD-INTERSECT. In this situation, q_k denotes the first operator whose cardinality is minimal w.r.t. the cardinality of the remaining operators. In experiments with our query optimizer, we found out that a constant factor of $2/3$ is a good heuristics for the selectivity of the n -way Intersect operator.

Finally, rule CARD-DIFFERENCE illustrates the cardinality inference of the binary Difference operator that reuses the standard formula well-known from the relational context.

5 Empirical Evaluation

Finally, this section discusses the empirical evaluation of the inference rules. In this context, we are not focussing on the cardinality inferences alone. Instead, we are interested in their interplay with the cost-based query optimizer and its ability to derive scalable QEPs.

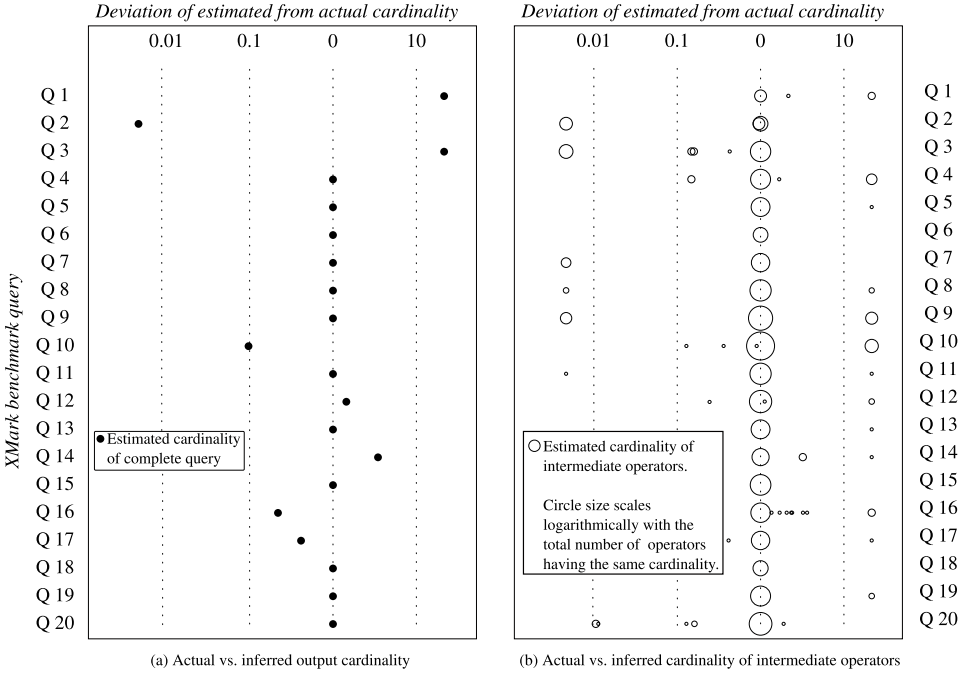


Figure 10: Actual vs. inferred cardinalities for the XMark benchmark queries

Our experiments were conducted on an Intel XEON quad core (3350) computer (2.66 GHz CPUs, 4 GB of main memory, 500 GB of external memory) running Linux with kernel version 2.6.14. Our native XDBMS server—implemented using Java version 1.6.0_07—was configured with a page size of 16 KB and a buffer size of 256 16-KB frames. The experimental results for the query execution times reflect the average values of five runs on a cold database buffer.

For the empirical evaluation, we used the XMark benchmark queries—a set of simple to complex XQuery expressions [SWK⁺02]—that serve very well to test the effectiveness and the ability of an XQuery processor to provide scalable QEPs. If not mentioned otherwise, we used an XMark document with scaling factor $f = 1.0$ that corresponds to an approximate document size of 110 MB.

In our first experiment, we used the cardinality inference rules to estimate the final output cardinality of each query and of each individual XQGM operator. Let us first have a

look at the results for estimated final output cardinalities. In Figure 10(a), we recorded the deviation of the estimated value from the actual cardinality on the x-axis. If there is a filled circle drawn at position 0, this means that there was no deviation between the estimated and the actual cardinality at all. On the other hand, if a circle is drawn at 10 or 0.1, then the estimated cardinality became victim of a 10-fold overestimate or a 10-fold underestimate, respectively. All circles depicted at the left-hand side of 0.01 and on the right-hand of 10.0 summarize overestimates and underestimates beyond these limits.

For 14 out of 20 queries, the cardinality estimates are close or equal to the actual cardinality. For queries Q 1–Q 3, the final cardinality estimates deviate significantly from the actual cardinality. For query Q 1, the selection predicate has a much lower selectivity than estimated using the 10 % heuristics. This problem can be easily overcome using refined statistics on value distribution, e. g., histograms. For queries Q 2 and Q 3, the selectivity of the positional predicates was not estimated correctly. Unfortunately, the error was propagated up to the estimate of the final query result. Nevertheless, the cardinalities for all performance-critical operators, such as access paths and SJs were estimated correctly. Therefore, the ability of the query optimizer to provide scalable QEPs is not affected in these situations—as indicated by Figure 11.

Our second experiment looked at the deviation between estimated and actual cardinalities for individual XQGM operators (Figure 10b). In Figure 10(a), all circles have the same size. In contrast, in Figure 10(b), we use the same notation as in Figure 10(a), except the fact that the circles are not filled anymore and their size is scaled logarithmically w. r. t. the total number of operators that have the same deviation ratio. For example, for the majority of operators in query Q 10, the inference rules estimated the correct output cardinality (in fact, 88 % of the inferred cardinalities were correct), hence, the largest circle is drawn at position 0. Moreover, the tiny circles between 0.1 and 0 and the medium-sized circle on the right-hand side of position 10 indicate outliers, from which the estimation rules were able to recover successfully. In total, for almost all queries, the inference rules provided for the majority of operators exact cardinality estimates. Even though the inference rules produced some outliers, they have little effect on the shape of the final QEP, because they mostly are related to GroupBy and Unnest operators that cannot be removed and where no alternative implementation exists.

Our third experiment shows that the inference rules are robust enough to support the query optimizer in deriving scalable query plans. Figure 11 depicts the execution times of the XMark benchmark queries on different document sizes (ranging from 110 KB to 1.1 GB). Besides the selection of implementations and SJ reordering, the cost-based query optimizer [WHdS10] selected indexes based on the recommendations of XTC’s auto-indexing feature [SH10]. For these tasks, reliable cardinality information is crucial to derive sufficiently efficient QEPs.

The execution times of most queries scale linearly with the document size. For small documents (size ≤ 10 MB), the average scale factor is even at most 6.85, i. e., an increase of the document size by factor 10 results only in a 6.85 times longer execution time. For the largest document in our experiment (1.1 GB), we still get an average scale factor of 10.5 for all queries except of Q 11 and Q 12. In contrast, queries Q 11 and Q 12 are very complex, include non-selective joins, and produce very large intermediate results that

scale quadratically with the document size. Therefore, the execution time of optimal plans increases quadratically, too. Hence, this is not an error of the optimizer but simply reflects the document and query characteristics.

Let us once again emphasize the importance of reliable cardinality estimates for the evaluation of value-based joins. Using the inference rules, e. g., for query Q9, the optimizer was able to propose a plan that was almost 18 times faster than without refined cardinality information. Though, it is noteworthy how query optimization time (from query parsing to the generation of the physical plan) relates to the overall execution time: On average, 97.62 % of the time is spent for query execution and only 1.54 % of the time was consumed by query optimization. Hence, cardinality inference has only a low impact on optimization time. As indicated by earlier experiments, non-optimized QEPs are up to two orders of magnitude slower than their optimized counterparts. Consequently, it is worth spending this small amount of time to get significantly better results.

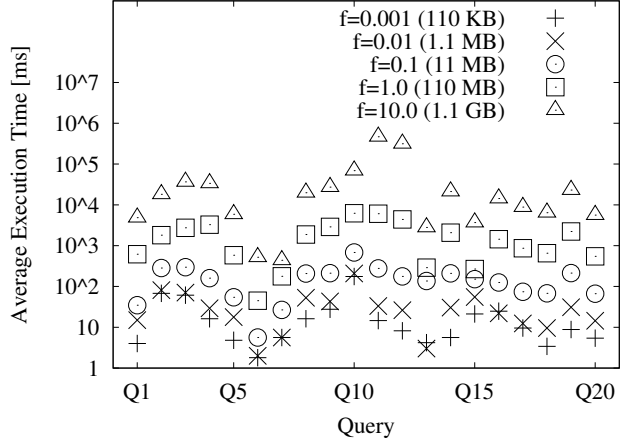


Figure 11: Scalability on XMark benchmark queries

On average, 97.62 % of the time is spent for query execution and only 1.54 % of the time was consumed by query optimization. Hence, cardinality inference has only a low impact on optimization time. As indicated by earlier experiments, non-optimized QEPs are up to two orders of magnitude slower than their optimized counterparts. Consequently, it is worth spending this small amount of time to get significantly better results.

6 Conclusions and Future Work

In this paper, we introduced and experimentally evaluated a set of inference rules that allow for effective cardinality estimation in native XDBMSs. To the best of our knowledge, this is the first approach that enables more precise cardinality estimation in such systems. Moreover, we have discussed how we can use the inference rules to provide termination criteria for cost-based query unnesting and support an appropriate selection of value-based join algorithms. In combination with a rich set of rewrite rules [WHdS10] and our generic index mechanism [HMB⁺10], the inference rules provide the foundation for XTC's cost-based query optimizer and enable it to derive scalable QEPs for a wide range of XQuery expressions. Though our experimental results are promising, there is still room for optimization. By refining the generalized 10 % heuristics and by focussing on a more precise treatment of positional predicates, we expect further improvements in estimation accuracy.

References

- [AAN01] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proc. VLDB Conf.*, pages 591–600, 2001.

- [AH08] José de Aguiar Moraes Filho and Theo Härder. EXsum—An XML Summarization Framework. In *Proc. IDEAS*, pages 139–148, 2008.
- [AKJP⁺02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE Conf.*, pages 141–154, 2002.
- [BEH⁺06] Andrey Balmin, Tom Eliaz, John Hornibrook, et al. Cost-Based Optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–320, 2006.
- [BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD Conf.*, pages 310–321, 2002.
- [FHK⁺02] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, et al. Anatomy of a Native XML Base Management System. *VLDB Journal*, 11(4):292–314, 2002.
- [FHR⁺02] Juliana Freire, Jayant R. Haritsa, Maya Ramanath, et al. StatiX: Making XML Count. In *Proc. SIGMOD Conf.*, pages 181–191, 2002.
- [FM07] Damien K. Fisher and Sebastian Maneth. Structural Selectivity Estimation for XML Documents. In *Proc. ICDE Conf.*, pages 626–635, 2007.
- [FMM⁺07] Mary Fernández, Ashok Malhotra, Jonathan Marsh, et al. XQuery 1.0 and XPath 2.0 Data Model (XDM)—W3C Recommendation 23 January 2007. <http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/>, 2007.
- [GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB Conf.*, pages 436–445, 1997.
- [HMB⁺10] Theo Härder, Christian Mathis, Sebastian Bächle, et al. Essential Performance Drivers in Native XML DBMSs. In *Proc. SOFSEM Conf.*, volume 5901 of *LNCS*, pages 29–46, 2010.
- [JAKC⁺02] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, et al. TIMBER: A Native XML Database. *VLDB Journal*, 11(4):274–291, 2002.
- [MAG⁺97] Jason McHugh, Serge Abiteboul, Roy Goldman, et al. Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, 26(3):54–66, 1997.
- [Mat07] Christian Mathis. Integrating Structural Joins into a Tuple-Based XPath Algebra. In *Proc. BTW*, volume 103 of *LNI*, pages 242–261, 2007.
- [Mat09] Christian Mathis. *Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems*. PhD thesis, University of Kaiserslautern, Germany, 2009.
- [PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD Conf.*, pages 39–48, 1992.
- [SAC⁺79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, et al. Access Path Selection in a Relational Database Management System. In *Proc. SIGMOD Conf.*, pages 23–34, 1979.
- [Sar03] Carlo Sartiani. A General Framework for Estimating XML Query Cardinality. In *Proc. DBPL*, pages 257–277, 2003.
- [SH10] Karsten Schmidt and Theo Härder. On The Use of Query-Driven XML Auto-Indexing. In *Proc. ICDE SMDb Workshop*, 2010.
- [SWK⁺02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, et al. XMark: A Benchmark for XML Data Management. In *Proc. VLDB Conf.*, pages 974–985, 2002.
- [TGMS08] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. Dependable Cardinality Forecasts for XQuery. *Proc. VLDB Endowment*, 1(1):463–477, 2008.
- [WHdS10] Andreas M. Weiner, Theo Härder, and Renato Oliveira da Silva. Visualizing Cost-Based XQuery Optimization. In *Proc. ICDE Conf.*, pages 1165–1168, 2010.
- [ZÖAI06] Ning Zhang, M. Tamer Özsu, Ashraf Aboulmaga, and Ihab F. Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. In *Proc. ICDE*, page 61, 2006.

A XQGM Components













































Operators		Intra-Operator Components	
	Select Operator		Projection Specification
	Document Access Operator		Sorting Specification
	Access Operator		Predicate
	Set Operator		Merge Specification
	Split Operator		Nesting Specification
	Merge Operator		Twig Specification
	Group-By Operator		Tuple Variable
	Unnest Operator		Outer Tuple Variable
	Arbitrary Operator (only used in rewrite patterns)		Dependent Tuple Variable
	Structural Join Operator		
	Tuple Variable Reference		
	Root Operator		
Expressions		Miscellaneous Components	
	Function Call		Structural Predicate
	Node Constructor		Projection Combination
	Literal		Sorting Combination
	Node Test		Distinct-Doc-Order
	Sequence Expression		Context Position Generation
	Range Expression		Context Size Generation
	Arithmetic Expression		Twig Node
	Boolean Expression		Grouping Twig Node
	Comparison Expression		Twig Node with Context Position
	Structural Predicate		
	Between Expression		
	Output Expression		
	Filter Expression		
	Positional Predicate		

Figure 12: Overview XQGM components [Mat09]

Efficient In-Memory Indexing with Generalized Prefix Trees

Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer,
Dirk Habich, Wolfgang Lehner
TU Dresden; Database Technology Group; Dresden, Germany

Abstract: Efficient data structures for in-memory indexing gain in importance due to (1) the exponentially increasing amount of data, (2) the growing main-memory capacity, and (3) the gap between main-memory and CPU speed. In consequence, there are high performance demands for in-memory data structures. Such index structures are used—with minor changes—as primary or secondary indices in almost every DBMS. Typically, tree-based or hash-based structures are used, while structures based on prefix-trees (tries) are neglected in this context. For tree-based and hash-based structures, the major disadvantages are inherently caused by the need for reorganization and key comparisons. In contrast, the major disadvantage of trie-based structures in terms of high memory consumption (created and accessed nodes) could be improved. In this paper, we argue for reconsidering prefix trees as in-memory index structures and we present the *generalized trie*, which is a prefix tree with variable prefix length for indexing arbitrary data types of fixed or variable length. The variable prefix length enables the adjustment of the trie height and its memory consumption. Further, we introduce concepts for reducing the number of created and accessed trie levels. This trie is order-preserving and has deterministic trie paths for keys, and hence, it does not require any dynamic reorganization or key comparisons. Finally, the *generalized trie* yields improvements compared to existing in-memory index structures, especially for skewed data. In conclusion, the *generalized trie* is applicable as general-purpose in-memory index structure in many different OLTP or hybrid (OLTP and OLAP) data management systems that require balanced read/write performance.

1 Introduction

Index structures are core components of typical data management systems. While this area has been studied for a long time, many aspects need to be reconsidered in the context of modern hardware architectures. Due to the increasing main-memory capacity and the growing gap between CPU speed and main-memory latency [MBK00], especially, in-memory indexing gains in importance. The specific characteristics of in-memory indexing compared to disk-based indexing are that (1) pointer-intensive index structures with small node sizes can be preferred instead of page-based structures due to smaller block granularities of main memory, and that (2) the number of required key transformations and comparisons as well as efficient main memory management and cache consciousness are crucial influencing factors on the overall performance. For update-intensive in-memory indexing in the context of online transaction processing (OLTP), typically, tree-based or hash-based techniques are used, while tries are usually neglected.

All of those structures have their specific drawbacks. Tree-based structures require re-

organization (tree balancing by rotation or node splitting/merging) and many key comparisons compared to hash-based or trie-based techniques. Hash-based techniques heavily rely on assumptions about the data distribution of keys and they require reorganization (re-hashing) as well. Tries are typically only designed for string operations, they often require dynamic reorganization (prefix splits), and they can cause higher memory consumption compared to tree- or hash-based structures. The disadvantages of tree-based and hash-based techniques are caused inherently by their structure. In contrast, the disadvantages of tries can be addressed appropriately. This optimization potential enables us to generalize existing trie structures in order to make them applicable for efficient in-memory indexing.

While tree-based and hash-based structures still have their application areas, we argue that a generalization of existing trie structures for in-memory indexing can, in particular with regard to a balanced read/write performance and for skewed data, achieve performance improvements compared to existing index structures. This trie generalization focuses on the goals of (1) trie-indexing for arbitrary data types, (2) order-preserving key storage, (3) deterministic trie paths (no need for dynamic reorganization, except leaf splits), and (4) efficient memory organization. Our primary contribution is the reconsideration and adaptation of prefix trees for efficient in-memory indexing. Furthermore, we make the following more concrete contributions, which also reflect the structure of this paper:

- First of all, in Section 2, we survey array-based, tree-based, hash-based and trie-based index structures and discuss their main drawbacks.
- Then, in Section 3, we describe the generalization of prefix trees. This includes the formal foundation and a physical realization of the *generalized trie*.
- Subsequently, in Section 4, we introduce the optimization techniques *bypass jumper array* and *trie expansion*. Furthermore, we provide insights into important read and write algorithms as well as additional memory optimization concepts.
- We present selected results of our experimental evaluation in Section 5.
- Finally, we conclude the paper and mention future work in Section 6.

2 Analysis of Existing Solutions

In order to give an overview of related index structures, we briefly survey the main categories of in-memory index structures but refer for more details to a comparison of basic in-memory structures [LC86] and a detailed time analysis, which also includes the number of required key comparisons [LDW09].

As the notation, we use the set of records R , where N denotes the number of records, with $N = |R|$. Each record $r_i \in R$ exhibits the structure $r_i = (k_i, \alpha_i)$, where k_i denotes the key and α_i denotes the associated information (payload). Furthermore, all records of R exhibit the same data type, where k with $k = |k_i|$ denotes the length of all keys in terms of the number of bits. If data types of variable length are used, the key length denotes the maximum key length. For example, a `VARCHAR(256)` results in $k = 2,048$.

Essentially, index structures are distinguished into the categories: (1) sorted arrays, (2) trees, (3) hash-based structures, and (4) prefix-trees (tries). In the following, we survey existing work according to these categories.

Sorted Arrays. The simplest index structure is an array of records. Unsorted arrays cause linear time complexity of $O(N)$. Hence, often sorted arrays are used in combination with *binary search* [Knu97]. It is known that the worst-case time complexity of binary search is $O(\log N)$. We have recently presented a k -ary search algorithm [SGL09] that uses SIMD instructions, yielding a significant improvement over binary search. However, the worst-case complexity is still $O(\log N)$. Although sorted arrays are advantageous for several application scenarios, they fall short on update-intensive workloads due to the need for maintaining the sorted order of records (moving records). In case of partitions of many duplicates, the shuffling technique [IKM07] (that moves as few as possible records of a partition) can minimize the costs for order maintenance. Another approach is to use sorted arrays with gaps according to a used fill factor such that only few tuples might be moved but the space requirements are increased. While a linked list would allow for more efficient updates, it is not applicable for binary search.

Tree-Based Structures. Typically, tree-based structures are used for efficient in-memory indexing. These structures are distinguished into unbalanced and balanced trees. The *binary tree* [Knu97] is an unbalanced tree where a node has at most two children. This structure can degenerate, causing more nodes to be accessed than for a balanced tree. In contrast, especially for in-memory indexing, balanced tree-structures are used. There is a wide variety of existing structures such as *B-trees* [BM72], *B⁺-trees* [Jan95], *red-black-trees* [GS78], *AVL-trees* [Knu97] and *T-trees* [LC86]. With the aim of tree balancing, rules for node splitting/merging or tree rotation are used. Current research focuses on cache-conscious tree-structures for T-Trees [LSLC07] and B-Trees [CGMV02, RR99, RR00] and on exploiting modern hardware like SIMD instructions [ZR02] or architecture-aware tree indexing on CPUs and GPUs [KCS⁺10]. All of those balanced tree structures exhibit a logarithmic time complexity of $O(\log N)$ in terms of accessed nodes for all operations. Additionally, they require a total number of $O(\log N)$ key comparisons. This is especially important when indexing arbitrary data types such as VARCHAR.

Hash-Based Structures. In contrast to tree-based structures, hash-based structures rely on a hash function to determine the slot of a key within the hash table (an array). Depending on the used hash-function, this approach makes assumptions about the data distribution of keys and can be order-preserving. For *chained bucket hashing* [Knu97], no reorganization is required because the size of the hash table is fixed. However, in the worst case, it degenerates to a linked list and thus, the worst-case search time complexity is $O(N)$. In contrast to this, there are several techniques that rely on dynamic reorganization such as *extendible hashing* [FNPS79], *linear hashing* [Lit80, Lar88], and *modified linear hashing* [LC86]. Current research focuses on efficiently probing multiple hash buckets using SIMD instructions [Ros07]. Due to reorganization, those structures exhibit a search time complexity of $O(1)$. However, additional overhead for dynamic hash table extension and re-hashing (worst case: $O(N)$) is required and thus it can be slower than tree-based structures. In conclusion, hash-based structures are advantageous for uniformly distributed keys rather than for skewed data.

Trie-Based Structures. The basic concept of prefix trees (tries) [Fre60]—also called digital search trees [Knu97]—is to use parts of a key k_i (with key length k) to determine the path within the trie. Prefix trees are mainly used for string indexing, where each node holds an array of references according to the used character alphabet [HZW02] and therefore they were neglected for in-memory indexing of arbitrary data types [LC86]. With that concept, the worst-case time complexity is $O(k)$ for all operations because, in general, the number of accessed nodes is independent of the number of records N (independent of the trie fill factor). However, the maximum number of indexable records is $N' = 2^k$. Compared to tree-based structures, more nodes are accessed because $k \geq \log N$, where $k = \log N$ only if $N' = N$. However, only partial key comparisons are required per node, where in the worst case, a single full key comparison is required in total for any operations. The most recognized trie structure is the *radix tree* [Knu97], where *radix tree*, *patricia trie* [Mor68] and *crit-bit tree* (critical bit) are used as synonyms. Essentially, these structures maintain a tree of string prefixes. Each node represents the string position, where the strings of the left and the right sub-trie differ. Thus, a node has at least two children and each edge can encode multiple characters [Mor68]. In contrast to traditional tries (using one character per trie level), this yields a significant string compression. Typically, this structure is only used for string indexing and efficient string operations. This data type restriction was addressed with the extension *Kart* (key alteration radix tree), where bit positions, rather than single character positions, are used for difference encoding. This allows for indexing arbitrary data types but due to the encoding of different prefix lengths within a trie, there is still the need for reorganization (arbitrary prefix splits). In conclusion, the advantage of trie structures is the good time complexity of $O(k)$ with only few (partial) key comparisons, which is especially useful when indexing strings. Existing trie-based structures are therefore mainly designed for those string operations rather than for indexing arbitrary data types.

Hybrid Structures. Furthermore, there are several hybrid index structures such as *prefix hash trees* [CE70] (trie and hash), *HAT-tries* [AS07] (trie and hash), *ternary search trees* [BS97] (trie and binary tree), *prefix B-trees* [BU77] (trie and B-tree), *partial keys* [BMR01] (trie and B-tree/T-tree), *J^+ -trees* [LDW09] (trie and B-tree/T-tree), *CS-prefix-trees* [BHF09] (trie and CSB-tree), and *Burst-tries* [HZW02] (trie and arbitrary structure for containers). Interestingly, all of these here mentioned hybrid structures use to some extend trie-based concepts.

3 Generalized Prefix Trees

Due to the disadvantages of existing structures, we outline our so-called *generalized trie* as a new in-memory data structure. It is a generalization of prefix trees (tries) for indexing arbitrary data types of fixed and variable length in the form of byte sequences. The novel characteristic compared to existing trie-based structures is an assembly of known and some new techniques. In detail, we use (1) a prefix size of variable length, (2) a bypass structure for leading zeros, (3) dynamic, prefix-based trie expansion, and (4) optimizations for pointer-intensive in-memory indexing. In this section, we focus on the formal foundation of this trie generalization, the core operational concepts, and still existing problems.

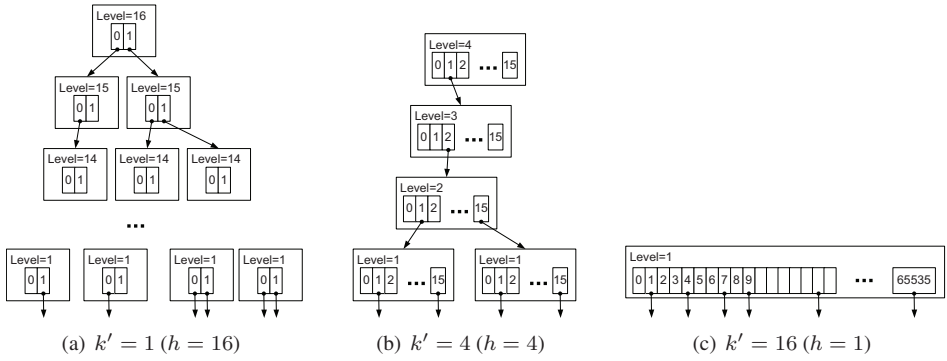


Figure 1: Example Generalized Trie with $k = 16$

3.1 Formal Foundation

As the foundation of this work, we define the *generalized trie* as follows:

Definition 1 Generalized Trie: *The generalized trie is a prefix tree with variable prefix length of k' bits. We define that (1) $k' = 2^i$, where $i \in \mathbb{Z}^+$, and (2) k' must be a divisor of the maximum key length k . The generalized trie is then defined as follows:*

- Given an arbitrary data type of length k and a prefix length k' , the trie exhibits a fixed height $h = k/k'$.
- Each node of the trie includes an array of $s = 2^{k'}$ pointers (node size).

The generalized trie is a non-clustered index with h levels. The root node describes level h , while nodes of level 1 are leaf nodes and point to the data items. Thus, leaf nodes are identified by the trie structure. The trie path for a given key k_i at level l is determined with the $(h - l)^{th}$ prefix of k_i . The single nodes of the trie are only created if necessary.

As a result, we have a deterministic prefix tree that indexes distinct keys of arbitrary data types without the need for (1) multiple key comparisons or (2) reorganization. The prefix length k' can be used to adjust the required space of single nodes and the number of accessed nodes for an operation. The following example shows the resulting spectrum.

Example 1 Configuration of Generalized Tries: Assume the data type *SHORT* (2) with a key length of $k = 16$. On the one side, we can set $k' = 1$, where each node contains two pointers (bit set/bit not set). This would result in a structure similar to a binary tree with a trie height $h = 16$. On the other side, we could set $k' = 16$, which results in a height $h = 1$ node containing an array of 65,536 pointers to keys. In the latter case, the complete key determines the array index. Figure 1 illustrates this configuration possibility using $k' = 1$ (Figure 1(a), binary prefix tree), $k' = 4$ (Figure 1(b), well-balanced hierarchical configuration), and $k' = k = 16$ (Figure 1(c), a sorted array with gaps).

Clearly, when configuring k' , this is a trade-off between the number of accessed nodes and space consumption. An increasing k' will cause decreased node utilization and thus, in-

creased memory consumption, while the steps from the root to the leafs are reduced for all operations. Furthermore, the generalized trie has the following two important properties:

- *Deterministic Property*: Using a prefix length of k' , a single node contains $s = 2^{k'}$ pointers. Each key has then only one path within the trie that is independent of any other keys. Due to these deterministic trie paths, only a single key comparison and no dynamic reorganization are required for any operations.
- *Worst-Case Time Complexity*: Based on the given key type of length k , the generalized trie has a time complexity of $O(h)$ and hence, has constant complexity in terms of the fill factor of the trie (the number of tuples N). Due to the dynamic configuration of k' , we access at most $h = k/k'$ different nodes (where h however might be higher than $\log N$), and compare at most $O(1)$ keys for any operations.

In contrast to the original trie [Fre60], we use a variable prefix length (not single characters) and have a deterministic trie of fixed height. The fixed height allows the determination of leaves. If the root level is given by $h = k/k'$, leaves are only present at level 1.

For data types of variable length such as `VARCHAR`, the trie height is set to the maximum key length rounded up to a factor of k' . In order to ensure that leaves are only present at level 1, keys with length $k < h \cdot k'$ are *logically* padded with zeros at the end because padding at the beginning would lead to loosing the property of being order-preserving.

3.2 Physical Data Structure and Existing Problems

Based on the formal definition of the *generalized trie*, we now explain the `IXByte` that is a physical data structure realizing such a trie generalization. We describe the main data structure and its most important operational concepts.

The `IXByte` is designed as a hierarchical data structure in order to leverage the deterministic property of the generalized trie. A single index (`L0Item`) is created with the data type length as a parameter. In order to support also duplicates of keys, we use a further hierarchical structure of key partitions k_i (`L1Item`) and payloads α_i (`L2Items`). Then, the single keys are indexed as a generalized trie, while the payloads of all records associated with the same key are maintained as a list of `L2Items` within this `L1Item` partition. Furthermore, the node data structure used within the `IXByte` is defined as an array of $s = 2^{k'}$ `void*` pointers. We use generic `void*` pointers in order to refer to both inner nodes and `L1Items` (leaves), where the fixed maximum trie height h determines the leaves. The `IXByte` works on order-preserving byte sequences (big-endian) and therefore arbitrary data types can be supported as long as their keys can be transformed accordingly.

Example 2 `IXByte` Operations: Assume an index of data type `SHORT (2)` with $k = 16$ and a prefix length of $k' = 4$. We insert the record $r_i = (107, \text{value3})$, where the resulting key parts are illustrated at the left side of Figure 2 and by emphasized pointers. We start at the root node (level 4) and use the value of bits 0-3 as array index in order to determine the child pointer. We repeat this with bits 4-7 on level 3, and with bits 8-11 on level 2. Finally, on level 1, we use the value of bits 12-15 as our array index. We know that this

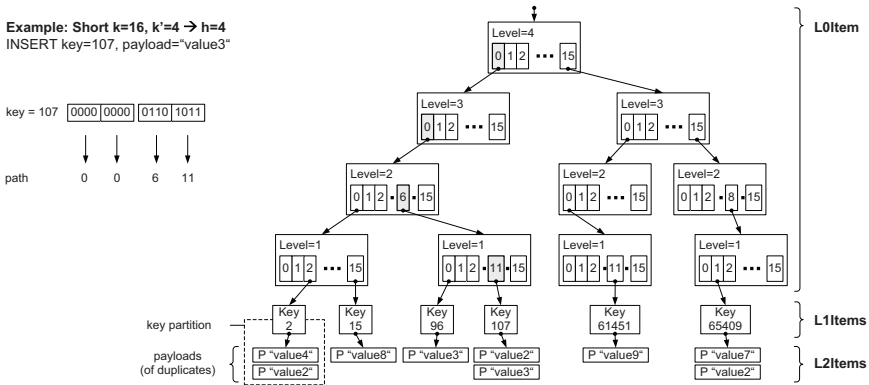


Figure 2: Example IXByte for $k = 16$, $k' = 4$, $h = 4$

pointer must reference an *L1Item* (key partition). If no partition exists, we create a new one; otherwise, we just insert the payload into the partition.

The *deterministic property*, where the trie path depends only on the key itself, implies that this structure is update-friendly because no reorganizations (prefix splits) are required. At the same time it also allows efficient lookups because no key comparisons are required. However, two typical problems arise when using such a trie structure:

Problem 1 Trie Height: A problem of this general trie solution is the height of the trie (number of levels). For example, a VARCHAR (256) index with $k' = 4$ would result in a trie height $h = 512$. In this scenario, keys with a variable key length k smaller than $h \cdot k'$ are logically padded with zeros at the end in order to ensure an order-preserving deterministic structure. The advantage of tries—in the sense of constant time for all operations—might be a disadvantage because the constant number of nodes to be accessed is really high (512). This problem has two facets, namely the number of accessed trie nodes (operational) and the number of created nodes (memory consumption).

Problem 2 Memory Consumption: Due to the fixed trie height, an inserted key can cause the full expansion (creation of trie nodes) of all levels of the trie if no prefix can be reused. In the worst case, an insert causes the creation of $h - 1$ nodes. Furthermore, each node requires a space of $2^{k'} \cdot \text{size}(\text{ptr})$ byte. For example, on a 64bit architecture, where a pointer ptr requires 8 byte, and with a prefix length of $k' = 4$, the node size is 128 byte.

In the following, we tackle these problems. The trie height is reduced by the techniques *trie expansion* and *bypass jumper array*. There, *trie expansion* implicitly leads to decreased memory consumption as well. In addition, we also apply an explicit memory reduction by *pointer reduction* and *memory alignment* (structure compression).

4 Selected Optimization Techniques and Algorithms

In this section, we present selected optimization techniques that address the problems of the potentially large trie height and memory consumption. We also discuss selected algorithms.

4.1 Bypass Jumper Array

Approaches that reduce the large height of the trie (e.g., $h = 512$ for $k = 2048$ and $k' = 4$) are required, while preserving the properties of the generalized trie.

The core concept of the technique *bypass jumper array* is to bypass trie nodes for leading zeros of a key. In order to enable this, we (1) preallocate all direct 0-pointers (nodes where all parents are only referenced by 0-pointers) of the complete trie, and we (2) create a so-called jumper-array of size h , where each cell points to one of those preallocated nodes. Finally, we (3) use the jumper array to bypass higher trie levels if possible. We show the concept of this *bypass jumper array* using our running example:

Example 3 Bypass Jumper Array: Recall Example 2. For this index of height $h = 4$ (Figure 3), we (1) preallocate all four direct 0-pointers, (2) create the jumper array c of size four, where cell c_j corresponds to the trie level l with $j = l$. Then, we (3) use the jumper array to jump directly to level $l = \lceil |k_i|/k' \rceil$ that is determined by counting leading zeros (clz). For example, when inserting the key $k_i = 107$, we can directly jump to the preallocated node at level 2.

With regard to the applicability, we distinguish data types of fixed and variable length:

First, for data types with variable-length $|k_i|$ (e.g., VARCHAR), we change the index layout. Keys with a length smaller than $h \cdot k'$ are now *logically* padded with zeros at the beginning and thus, we lose the property of being order-preserving. However, if this is acceptable the benefit is significant. We assume that the key length $|k_i|$ is uniformly distributed in the interval $[1, k]$. As a result, the number of accessed trie nodes is reduced from h to $h/2$ in expectation. This causes significant performance improvements due to fewer accessed nodes (logical partitioning into h subtrees but with prefix sharing). However, the time complexity is still $O(h)$.

Second, this technique also works for fixed-length data types such as INT, by counting leading zeros and without any index layout modification. Unfortunately, for uniformly distributed keys, the impact is lower because the probability of having a key length $|k_i| < k$ depends on k' with $P(\lceil |k_i|/k' \rceil < k/k') = 1/2^{k'}$. For example, if $k' = 4$, only 1/16 of all keys would benefit at all from this technique. The probability of a specific length is given by $P(\lceil |k_i|/k' \rceil = k/k' - x) = (1/2^{k'})^x$, where x denotes the number of nodes that can be bypassed. However, numerical values are typically non-uniformly distributed and rather small (e.g., key sequences) and hence, they could also benefit from this technique.

In the description of algorithms, we will refer to this technique as `bypass_top_levels`.

4.2 Trie Expansion

Another reason for the huge trie height is that the index has a fixed height according to its data type and thus, each record is stored on level 1. This is also a reason for the huge memory consumption, because a single record can cause the creation of $h - 1$ new nodes.

Based on this problem, we investigated the dynamic *trie expansion*. The core concept is to defer the access to and creation of trie nodes during insert operations until it is required

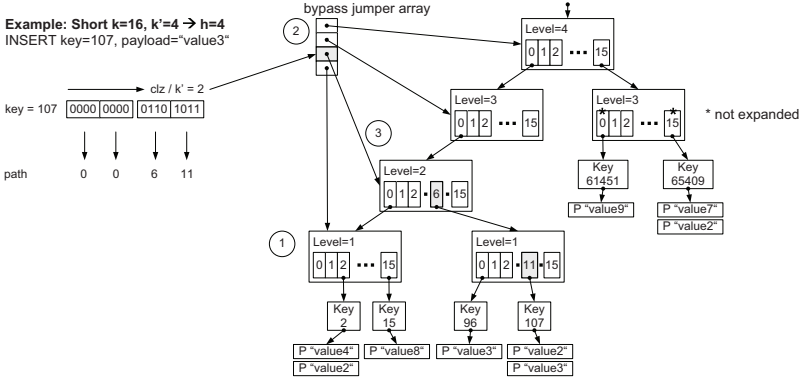


Figure 3: Example IXByte for $k = 16$, $k' = 4$, $h = 4$ with *bypass jumper array* and *trie expansion*

with respect to the *deterministic property*. In other words, a tuple can now be referenced on any level rather than only at the leaf nodes. It is possible to reference a record instead of an inner node if and only if there is only one record that exhibits a particular prefix (that is given by the position in the trie). A similar concept has already been used within the original algorithm [Fre60] for string indexing. For data types of variable length (e.g., VARCHAR), the logical padding with zeros at the end (or beginning, respectively) ensures the deterministic property even in case of trie expansion. We use the following example to illustrate the idea of *trie expansion*:

Example 4 Trie Expansion: Recall our running example of a *SHORT* (2) index with $h = 4$. Figure 3 includes an example of the trie expansion. Consider the keys $k_1 = 61,451$ and $k_2 = 65,409$. If we insert $k_2 = 65,409$, we can refer to it on level 4 instead of on level 1 because so far, no other key with a first prefix of 1111_2 exists. Further, if we insert $k_1 = 61,451$, we need to expand the third level for this sub-trie because k_1 and k_2 both share the same prefix 1111_2 . However, we do not need to expand any further because the second prefix of both keys (0000_2 and 1111_2) differs. This still ensures the deterministic property because we only expand the deterministic trie path of keys if it is required.

In contrast to the bypass jumper array, this technique does not only bypass existing nodes but influences the number of created nodes. While the dynamic *trie expansion* significantly reduces the trie height and therefore also the memory consumption for sparsely populated subtries, it requires changes of the trie node data structure. A node is now defined as an array of $s = 2^{k'} \text{ void}^*$ pointers and an array of $\lceil s/8 \rceil$ bytes to signal the expansion of a sub-trie. The i^{th} bit of this byte array determines if the i^{th} pointer references an inner node; otherwise, the pointer directly references a record or is NULL. This is required due to the generic pointers. However, the evaluation of this flag is a simple bit mask operation. In the algorithmic description, we refer to this as `isExpanded`. While so far, we have required $2^{k'} \cdot \text{size}(ptr)$ byte for a node, now a single node has a size of $\text{size}(node) = 2^{k'} \cdot \text{size}(ptr) + \lceil 2^{k'}/8 \rceil = 2^{k'} \cdot \text{size}(ptr) + \lceil 2^{k'-3} \rceil$. As an example, for $k' = 4$ without trie expansion, the node size was 128 byte; enabling trie expansion adds two more bytes to the node. The downside of this technique are costs for splitting and merging of nodes (but still no inner prefix splits) as well as an unaligned data structure

with the drawback that the node size is no longer a factor or divisor of the cache line size. However, due to the significant reduction of created nodes, the total memory consumption is significantly reduced, which improves the performance for all operations.

4.3 Memory Optimization

The solution presented so far still has the drawbacks of (1) creating many small data structures (many `malloc` calls), (2) a fairly high memory consumption, and (3) an unaligned data structure (expanded flags). Hence, we apply several memory optimization techniques.

First, we use *memory preallocation* in order to address the many small data structures. Each instance of an index includes a byte array of configurable size that is created using virtual memory upon index creation. We hold a pointer to this array (`mem_ptr`) and the allocation position `pos` as well as we use lists of free objects for reusing memory. We benefit because the operating system only maps physical pages of this array when they are accessed. Second, we reduce the memory consumption of trie nodes with the concept of *reduced pointers*, which store only the offset within the preallocated memory instead of the pointer to a certain memory position itself. The real pointer is then computed by the reference to the full array (`mem_ptr`) plus the given offset (reduced pointer), which reduces the required size of a pointer; for example (for a preallocated memory of less than 2 GB), from 8 byte to 4 byte on 64bit architectures. Third, cache-consciousness has significant influence on the overall performance. We align the trie node size to factors of the cache line size. Based on the concept of reduced pointers, the idea is to use the first bit of such a pointer as internal flag to indicate whether or not this pointer is an expanded node. Following this, we do not need any additional flags per node. As a result, the final structure of our trie node is defined as an array of $s = 2^{k'} \text{ uint}$ reduced pointers resulting in a node size of $\text{size}(\text{node}) = 2^{k'} \cdot \text{size}(\text{rptr}) = 2^{k'} \cdot 4$. For example, using $k' = 4$, we get a node size of 64 byte, which is equal to the cache line size of modern processors.

4.4 Algorithms

So far, we have mainly discussed structural properties of the `IXByte`; now, we focus on the operational aspects. The main operations are `get(key)` (point query), `getMin()`, `getNext(key1)` (scan), `insert(key, payload)`, and `delete(key, payload)`. Updates are represented by a delete/insert pair. It is worth mentioning that keys of arbitrary data types are converted into byte arrays such that all operations are only implemented once according to this byte representation. All algorithms to search and modify generalized tries include the optimization approaches *bypass jumper array* and *trie expansion*. We use the operations `get` and `insert` as illustrative example algorithms.

Algorithm 1 shows the `get` algorithm. A single index instance `ix` includes the root *trie* node and the level of this root node (given by $h = k/k'$). First, the *bypass jumper array* is used to jump directly to the trie node of level $\lceil |k_i|/k' \rceil$ if required (line 2). Here, *level* and *node* are passed by reference and set accordingly. Then, the `get` algorithm mainly comprises a `while` loop (lines 3-10). For each iteration, we go one level down the trie,

Algorithm 1 get (non-recursive)

Require: index ix , key k_i , key length $|k_i|$

```
1:  $node \leftarrow ix.trie, \quad level \leftarrow ix.level$ 
2:  $bypass\_top\_levels(node, level, |k_i|, ix)$ 
3: while  $(level \leftarrow level - 1) > 0$  do
4:    $pos \leftarrow computePosition(level, k_i, |k_i|)$ 
5:   if  $\neg isExpanded(node, pos)$  then
6:     if  $node.ptr[pos].key = k_i$  then
7:       return  $node.ptr[pos]$ 
8:     else
9:       return NULL // null if no key or different key
10:   $node \leftarrow node.ptr[pos]$  // go one level down
```

Algorithm 2 insert (non-recursive)

Require: index ix , key k_i , key length $|k_i|$

```
1:  $node \leftarrow ix.trie, \quad level \leftarrow ix.level$ 
2:  $bypass\_top\_levels(node, level, |k_i|, ix)$ 
3: while  $(level \leftarrow level - 1) > 0$  do
4:    $pos \leftarrow computePosition(level, k_i, |k_i|)$ 
5:   if  $\neg isExpanded(node, pos)$  then
6:     if  $node.ptr[pos] \neq \text{NULL}$  then
7:        $entry \leftarrow node.ptr[pos]$ 
8:       if  $entry.key = k_i$  then
9:         return  $entry$ 
10:    while true do // node prefix splitting
11:       $tmp \leftarrow createNode()$ 
12:       $node.ptr[pos1] \leftarrow tmp$ 
13:       $setExpanded(node, pos)$ 
14:       $node \leftarrow tmp, \quad level \leftarrow level - 1$ 
15:       $pos1 \leftarrow computePosition(level, entry.k_i, |k_i|)$ 
16:       $pos2 \leftarrow computePosition(level, k_i, |k_i|)$ 
17:      if  $pos1 \neq pos2$  then
18:         $node.ptr[pos1] \leftarrow entry$ 
19:        return  $node.ptr[pos2] \leftarrow createL1(k_i, |k_i|)$ 
20:      else
21:         $pos \leftarrow pos1$ 
22:    else
23:      return  $node.ptr[pos] \leftarrow createL1(k_i, |k_i|)$ 
24:   $node \leftarrow node.ptr[pos]$  // go one level down
```

where the loop terminates if $level = 0$. For each iteration, we first compute the position within the pointer array (`computePosition`, line 4). It is determined by the $(h - l)^{\text{th}}$ prefix of key k_i . If the pointer specified by pos is not expanded (see *trie expansion*), there must be a reference to a key, or the pointer is NULL. Hence, by checking for the right key (line 6), we could simply return this key partition or NULL. Otherwise (the pointer is expanded), there must be a reference to another trie node.

Similar to the `get` operation, Algorithm 2 uses the same core concept for the `insert` operation. After we have jumped to the lowest possible trie node in case it is applicable (line 2), the algorithm comprises a main `while` loop (lines 3-24). The structure of this loop is similar to the `get` operation. However, we maintain the dynamic *trie expansion* of sub-tries. If a pointer is not expanded and not `NULL`, we have reached a record. In case the keys are equivalent, we simply return the existing entry. Otherwise, we use an inner `while` loop (lines 10-21) to expand the sub-trie as long as it is required (leaf node splitting). Basically, we can stop splitting such nodes if we reach a level where the prefixes of both keys are different (lines 17-19). If the current pointer is already expanded, we follow this pointer and go one level down (line 24). Note that this splitting does not require any dynamic reorganization because we just go down the trie as long as it is required.

Both algorithms use the `computePosition` multiple times in order to compute the current prefix at each trie level. If many levels of the trie are expanded, it is advantageous to pre-compute all positions in advance in one run over the key.

The other algorithms work similar to the presented ones. For `getMin`, `getNext`, and `delete`, we additionally maintain a stack of parent nodes and current positions on each level in order to realize non-recursive algorithms. The `getNext` searches for the current key, and starting from this position, it returns the minimum key that is greater than this. Further, the `delete` searches a specific record and then deletes the given (key,value) pair and recursively collapse trie nodes if required.

The index can also store `NULL` keys, which is required if used as a secondary index. Therefore, an additional single key partition (`L1Item`) is referenced by the index instance. Furthermore, all algorithms include related `NULL` checks at the beginning.

4.5 Discussion of Related Work

The closest work compared with our approach is the *Prefix Hash Tree* [CE70], which was defined by Coffmann and Eve from a theoretical perspective. They use a hash function h_j to map the key k_i into a hash code b_i of fixed length j . Thereafter, the prefix hash tree uses prefix-based tree nodes that contain $s = 2^{k'}$ child references. In the case of the simplest hash function of $b_i = h_k(k_i) = k_i$, this is comparable to the generalized trie.

In contrast to this, we presented the *generalized trie* that does not rely on any hash function, i.e., it does not require related indirections for overflow lists of different keys and it can index keys of variable length. Furthermore, we explained the `IXByte` as a physical realization of the generalized trie including optimizations for the large trie height and memory consumption as well as the description of efficient algorithms using this data structure.

5 Experimental Evaluation

In this section, we present selected experimental results concerning the performance, scalability, memory consumption, and the comparison with existing index structures. We used synthetically generated data, a real data set as well as the MIT main-memory indexing

benchmark [Rei09]. In general, the evaluation shows the following results:

- *Data Types*: The `IXByte` achieves high performance for arbitrary data types, where an increasing key length causes only moderate overhead.
- *Skew Awareness*: It turns out that the `IXByte` is particularly appropriate for skewed data (sequence, real) because many keys share equal prefixes, while uniform data represents the worst case.
- *Prefix Length*: A variable prefix length (specific to our `IXByte`) of $k' = 4$ turned out to be most efficient and robust due to (1) a node size equal to the cacheline size (64 byte) and (2) a good trade-off between trie height and node utilization.
- *Comparison*: The `IXByte` shows improvements compared with a B^+ -tree and a T-tree on different data sets. Most importantly, it exhibits linear scaling on skewed data. Even on uniform data (worst case) it is comparable to a hash map.

As a result, the `IXByte` can be used as a general-purpose data structure because, especially for skewed in-memory data, it is more efficient than existing tree-based or hash-based solutions. The complete C source code (including all experiments) is available at wwwdb.inf.tu-dresden.de/dexter. Our core data structure `IXByte` is part of an overall index server, which is able to maintain an arbitrary number of indices (of different data types) in a multi-threaded fashion. Essentially, it implements the API, defined by the MIT main-memory benchmark [Rei09] including memory management, synchronization and transaction logging (transient UNDO-log). All reported results were measured from *outside* the index server—except the comparison with B^+ -trees.

5.1 Experimental Setting

As our test environment, we used a machine with a quad core Intel Core i7-920 processor (2.67GHz) and hyper-threading (two threads per core). It uses Fedora Core 14 (64bit) as operating system and 6GB of RAM are available. Furthermore, we used the GCC compiler with the following flags: `-O3 -fPIC -lpthread -combine`.

While the proposed `IXByte` is able to index arbitrary *data types*, here, we used only the types `SHORT` (4 byte), `INT` (8 byte) and `VARCHAR(128)` (128 byte). In order to evaluate the influence of skew and other data properties we used the following data sets:

- *Key Sequence* (best-case synthetic data): We generated a key sequence (highest skew) of N records with values $[1..N]$ in sorted order. For the data type `VARCHAR`, we convert this value into a string representation.
- *Uniform Data Distribution* (worst-case synthetic data): In opposite to the key sequence, we additionally generated N records with uniformly distributed keys with values $[1..2^k]$ in unsorted order. For `VARCHAR(128)`, we first determined a random length in the interval $[1..128]$. For each position, we then picked a random character out of the alphabet of 52 printable characters.
- *Real Data*: Aside from the synthetically generated data, we also used the DBLP data set [Ley09] as a real data set. In more detail, we indexed all 2,311,462 distinct key attributes of this data set as a `VARCHAR(128)` index instance.

We evaluated the different operations `insert`, `get`, `getNext` (scan), and `delete`. Further, we varied the prefix length with $k' \in \{1, 2, 4, 8\}$. In addition, we compared our generalized prefix trie against other index structures: We used an unoptimized B⁺-tree [Avi09] and optimized it (memory optimizations) similar to our index structure in order to ensure a fair comparison. Furthermore, we used the MIT benchmark as well as the optimized hash map and T-tree implementations from the winner (Clement Genzmer) and another finalist (Cagrı Balkesen) of the SIGMOD Programming Contest 2009.

5.2 Synthetically Generated Data

The first set of experiments uses the synthetically generated data sets *sequence* and *uniform*. We fixed a prefix length of $k' = 4$ as well as enabled *trie expansion* and the described memory optimizations. Then, we measured the execution time for the mentioned operations. Figure 4 shows the results of this set of experiments.

Experiment S.1: The first experiment uses the sequence data set. The goal was to evaluate the performance with an increasing number of records N . For each database size, this comprises N inserts, N point queries (`get`), a scan of N elements (`getNext`) as well as N deletes, where we used all keys of the generated sequence. We repeated this experiment for the three different data types. Figure 4 (first row) shows the measured results. First, `SHORT`(4) shows the best performance due to the lowest trie height ($h = 8$). Note that for sequences, the complete trie height is expanded. We further observe that `INT`(8) ($h = 16$) shows only a minor overhead, while for `VARCHAR` ($h = 256$), this overhead is higher. Those differences are caused by the trie height and additional memory requirements for the larger keys. From an operation type perspective, `delete` always has the worst performance, followed by `insert`, which is reasoned by memory management in addition to the search operations. Further, `getNext` is typically slower than `get`, because for a `getNext`, a similar point query is used in order to find the old key, and from there, it finds the next key, which was required by scans with concurrent updates. Interestingly, `getNext` is better than `get` for `VARCHAR` indices, which was reasoned by cache displacement on `get` due to the huge height for the `VARCHAR` indices. Furthermore, `get`, `insert`, and `delete` require additional overhead for key generation and copying of keys into the local memory of our index server. The technique *bypass jumper array* led to an improvement (not included in the figures) between 9% and 17% for the different operations. However, and most importantly, all operations show (1) a linear scaling according to the number of tuples (constant time for a single operation) due to the fixed number of accessed nodes and (2) a moderate overhead according to the key size.

Experiment S.2: For a second experiment, we repeated Experiment S.1 with the *uniform* data set but exactly the same configuration. Figure 4 (second row) illustrates the measured results of this experiment. We observe similar characteristics—with some exceptions—compared to the *sequence* data set. In general, the index is notably slower for uniform key distributions. This has different reasons. Recall that uniform data is the worst case regarding space requirements, while this is the best case with respect to the average number of accessed nodes per operation. The costs for allocating/loading more memory are higher than the benefit reached by the lower number of trie nodes. Due to uniformly generated

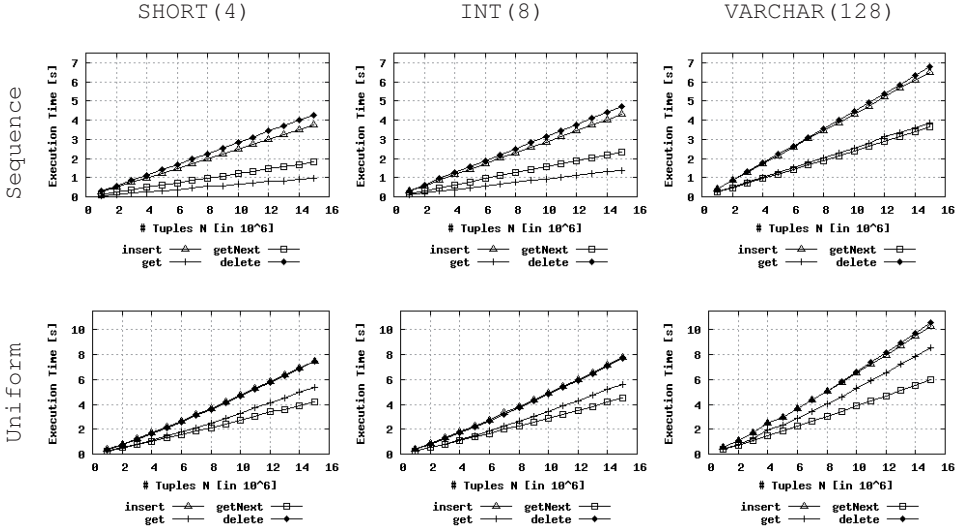


Figure 4: Basic Performance Results

keys, each single operation (except the scan operation) accesses different nodes. As a result of uniform access and the higher memory consumption, many nodes are emitted faster from the cache. Another difference of uniform keys to key sequences is that the execution time increases slightly super-linear with an increasing number of tuples (logarithmic time for a single operation). This effect is caused by the logarithmically increasing number of accessed nodes with an increasing number of tuples. The second difference concerns the scan performance, where the `getNext` is faster than `get` for all data types. Essentially, this is a caching effect because the scan is the only operation that accesses the records in sequential order. Hence, the higher trie nodes are cached across multiple operations. The technique *bypass jumper array* lead to an improvement (not shown in the figures) between 4% and 7% for the different operations, which is lower than in experiment S.1 because the trie does not use the full height for uniform data and thus fewer nodes are bypassed.

As a result, our index shows good performance and an almost linear scaling for sequences (best-case) and uniform data (worst case). However, it is best suited for skewed data.

5.3 Real DBLP Data

The *sequence* and *uniform* data sets are extremes in terms of skew and thus give upper and lower bounds for the number of accessed nodes as well as required space. We now additionally use the real-world DBLP data set that lies within this spectrum.

Experiment R.1: As a preprocessing step, we extracted all distinct publication keys, which are concatenations of the type of publication, the conference/journal name, and the short bibtex key (e.g., `conf/sigmod/LarsonLZZ07`). We used a `VARCHAR(128)` index and truncated all keys that exceeded this key length (only a few). The number of distinct items was 2,311,462. Then, we inserted this data set (in sorted order) and evaluated

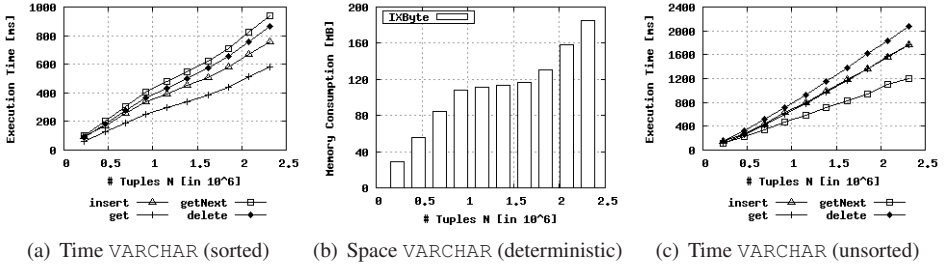


Figure 5: Performance on Real Data Set (DBLP)

execution time and space requirements. Figures 5(a) and 5(b) show the results. We observe an almost linear scalability with increasing data size, similar to our other experiments. However, in contrast to sequences and uniform keys, there are two main differences. First, the `getNext` operation shows the worst performance compared to all other operations, while in the other experiments, it was usually faster than `insert` and `delete`, because many prefixes are shared such that `insert` and `delete` become more efficient. Similar to this, `get` is also slower than in other cases compared to the `insert/delete` functions. This effect is caused by long keys, where we need to access many nodes for each point query. Note that due to transactional requirements each `getNext` also includes such a point query, which reasoned that `getNext` was the slowest operation. Furthermore, we see three main parts. Up to 700,000 records, there is a linear scaling. Then, from 700,000 to 1,600,000 records, we see a slower increase. From there to the end, we observe a scaling similar to the first part. This is caused by the huge number of conferences that all share the same prefix (e.g., `conf/sigmod/`), which results in good compression. It is also worth to note that the performance is strongly correlated to the required memory.

Experiment R.2: As a comparison, we repeated the experiment with unsorted keys (see Figure 5(c)). Due to the deterministic property, the total memory consumption was equivalent to the previous experiment. In contrast to sorted keys, we observe that the performance was much lower due to cache displacement because created nodes are distributed over the memory. The `getNext` operation now performs best because it is the only one with a sequential access pattern with regard to the trie nodes. Most important, we still observe a linear scaling due to the good compression.

As a result, the trie works also well for real-world data. We conclude that the higher the skew, the higher the performance because we require less memory.

5.4 Comparison with Existing Structures

For comparison with existing structures, we used a general-purpose B^+ -tree. We optimized the implementation with memory preallocation in order to achieve a fair comparison. For the `IxByte`, we used a prefix length of $k' = 4$, while we configured the B^+ -Tree as a tree of order four (pointers per node [Knu97]) because experiments with different configurations showed that this leads to highest performance for this experimental setting.

Experiment C.1: Due to the data type restriction of the used B⁺-tree, we used SHORT(4) only. We varied the number of tuples N and measured the execution time for insert and get as well as the memory consumption. The delete operation of the B⁺-tree was similar to the insert and a scan operation was not supported. We first used the *sequence* data set. In contrast to our other experiments, we measured the time for the operations of the core index structures rather than from outside the index server because the B⁺-tree does not implement the MIT main-memory benchmark API. Figure 6 (left column) shows the results. We yield an execution time improvement of up to factor 5 due to key comparisons and node splitting/merging within the B⁺-tree. Interestingly, the IXByte also requires three times less memory than the B⁺-tree. However, this is the best case for the prefix trie.

Experiment C.2: In addition, we repeated the experiment with the *uniform* data set. Figure 6 (right column) shows the results. The IXByte is still three times faster than the B⁺-tree. Both index structures are slower for this *uniform* data set. Due to different memory allocation behavior (uniform data is the worst case for IXByte), the B⁺-tree required only 50% of the memory that was required by IXByte.

Most importantly, the IXByte shows a linear scalability on skewed data with increasing number of tuples, while the B⁺-tree shows a super-linear scalability for both data sets.

5.5 MIT Benchmark

With regard to repeatability, we provide the results of the MIT main-memory indexing benchmark [Rei09] that was designed during the SIGMOD Programming Contest 2009 (see Figure 7(c) for a comparison with the winning implementation, where the uniform data distribution is the best case for the hash map, while it is the worst case for our IXByte). The benchmark is designed with uniform key distributions but there are also test cases for skewed data—namely the test cases of varying low bits (VLB) and varying high bits (VHB). In general, the benchmark creates a random set of indices (of types SHORT(4), INT(8), and VARCHAR(128)) and runs a transaction workload in a concurrent fashion, where each transaction comprises between 5 and 200 single operations (point queries, range queries, inserts and deletes). With regard to the repeatability of the

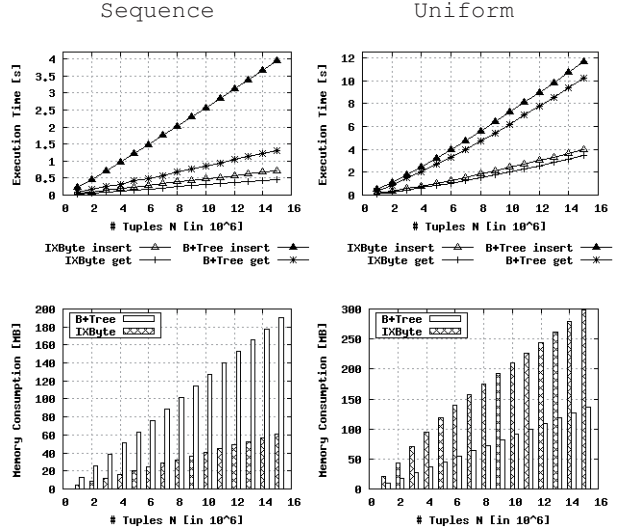


Figure 6: SHORT(4) Comparison IXByte and B⁺-Tree

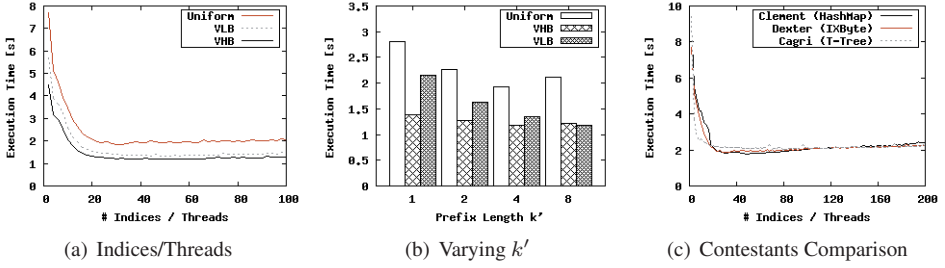


Figure 7: MIT Main-Memory Indexing Benchmark

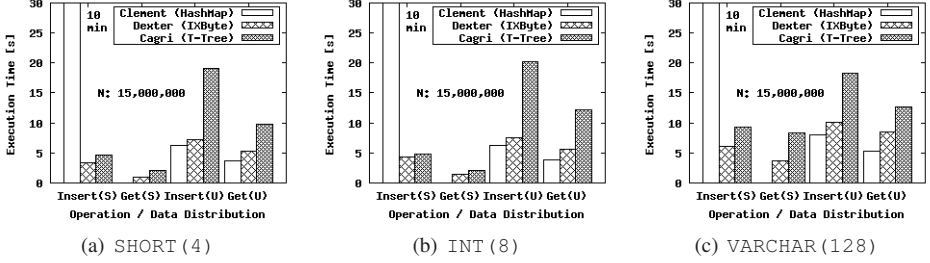


Figure 8: Contestants Comparison for Different Operations and Data Sets

obtained benchmark results, we used the same seed (1468) for all sub-experiments. We conducted a set of experiments varying the benchmark parameters, which include the number of indices and threads `NUM_IX`, the number of initial inserts per index `NUM_INSERTS`, and the total number of transactions `NUM_TX` over all indices. The results are shown in Figure 7, where we measured the overall benchmark execution times.

Experiment M.1: We varied the number of indices and threads, respectively. There, we fixed `NUM_INSERTS`=4,000, `NUM_TX`=800,000 (contest configuration) and measured the execution time for the three different data distributions. The results are shown in Figure 7(a). We observe that the index performed best on the VHB benchmark because there, the varied bits are represented by the first two node levels. In contrast, for VLB we fully expanded a long path because the varied bits are represented by the last two node levels. However, for both benchmarks, many keys are shared such that they both exhibit lower execution time than the uniform benchmark; the differences are moderate. Although our test machine has only eight hardware threads, our index server scales pretty well with an increasing number of threads. Note that we reached a CPU utilization of 800% (up from a number of 25 threads).

Experiment M.2: Beside the benchmark parameters, we also evaluated the influence of the prefix length k' on the execution time. We fixed `NUM_IX`=50, `NUM_INSERT`=4,000, and `NUM_TX`=800,000 and varied the prefix length $k' \in (1, 2, 4, 8)$. Figure 7(b) shows the results of this second sub-experiment. Essentially, for different data distributions, different prefix lengths are advantageous. For example, while for uniform data (and VHB), the best prefix length is $k' = 4$, for VLB it is $k' = 8$ because there the index is fully expanded for a certain key range. In general, for skewed data (e.g., *sequence*), larger prefix lengths are advantageous because certain subtrees are fully expanded.

Experiment M.3: Finally, we use the optimized hash map and T-tree implementations of the mentioned other contest finalists for a more detailed comparison regarding different operations (`insert`, `get`), data types (`SHORT(4)`, `INT(8)`, `VARCHAR(128)`) and data sets (sequence S and uniform U), where the results are shown in Figure 8. The hash map did not terminate within 10 min on the sequence data set such that these experiments were aborted. In contrast, we observe that our index performs best on sequence data, where the relative improvement over the T-tree is highest for `VARCHAR` due to the higher influence of the number of required key comparisons. On uniform data, the hash map performed best but our `IXByte` achieves only slightly worse performance and still outperforms the T-tree. In conclusions, the `IXByte` is especially beneficial in case of skewed data.

6 Conclusions

In-memory data structures are gaining importance but still exhibit certain disadvantages. While these are inherently given for tree-based and hash-based structures, we addressed the drawbacks of trie structures in order to make them applicable for in-memory indexing.

To summarize, we presented a generalized prefix tree (trie) for arbitrary data types that uses a variable prefix length. The major advantage is the deterministic property that eliminates the need for dynamic reorganization (prefix splits) and multiple key comparisons. Furthermore, we presented optimization techniques for reducing the number of accessed and created trie nodes (*bypass jumper array* and *trie expansion*). Additionally, we briefly discussed memory optimization techniques and efficient algorithms. In conclusion, the generalized trie is advantageous compared to existing solutions. The major benefit is an almost linear scalability with respect to the number of tuples, especially for skewed data.

Using this generalized trie, several further research issues arise, which include, for example, the (1) adaptive determination of the optimal prefix length k' during runtime, (2) a hybrid solution with different prefix lengths on different levels of a generalized trie (e.g., $\{16, 8, 4, 4\}$ for `SHORT(4)`), and (3) query processing on prefix trees, where database operations such as joins or set operations can exploit the deterministic trie paths.

References

- [AS07] Nikolas Askitis and Ranjan Sinha. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *ACSC*, 2007.
- [Avi09] Amittai Aviram. *B+ Tree Implementation*, 2009. <http://www.amittai.com/prose/bplus-tree.html>.
- [BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, 2009.
- [BM72] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1, 1972.
- [BMR01] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. Main-Memory Index Structures with Fixed-Size Partial Keys. In *SIGMOD*, 2001.
- [BS97] Jon Louis Bentley and Robert Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *SODA*, 1997.

- [BU77] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *ACM Trans. Database Syst.*, 2(1), 1977.
- [CE70] E. G. Coffman, Jr. and J. Eve. File structures using hashing functions. *Commun. ACM*, 13(7), 1970.
- [CGMV02] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B \pm Trees: optimizing both cache and disk performance. In *SIGMOD*, 2002.
- [FNPS79] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.*, 4(3), 1979.
- [Fre60] Edward Fredkin. Trie Memory. *Communications of the ACM*, 3(9), 1960.
- [GS78] Leonidas J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *FOCS*, 1978.
- [HZW02] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2), 2002.
- [IKM07] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *SIGMOD*, 2007.
- [Jan95] Jan Jannink. Implementing Deletion in B+-Trees. *SIGMOD Record*, 24(1), 1995.
- [KCS⁺10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, 2010.
- [Knu97] Donald E. Knuth. *The art of computer programming Volume I. Fundamental Algorithms, Third Edition*. Addison-Wesley, 1997.
- [Lar88] Per-Åke Larson. Linear Hashing with Separators - A Dynamic Hashing Scheme Achieving One-Access Retrieval. *ACM Trans. Database Syst.*, 13, 1988.
- [LC86] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *VLDB*, 1986.
- [LDW09] Hua Luan, Xiaoyong Du, and Shan Wang. Prefetching J⁺-Tree: A Cache-Optimized Main Memory Database Index Structure. *J. Comput. Sci. Technol.*, 24(4), 2009.
- [Ley09] Michael Ley. DBLP - Some Lessons Learned. In *VLDB*, 2009.
- [Lit80] Witold Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *VLDB*, 1980.
- [LSLC07] Ig Hoon Lee, Junho Shim, Sang Goo Lee, and Jonghoon Chun. CST-Trees: Cache Sensitive T-Trees. In *DASFAA*, 2007.
- [MBK00] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3), 2000.
- [Mor68] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4), 1968.
- [Rei09] Elizabeth G. Reid. Design and Evaluation of a Benchmark for Main Memory Transaction Processing Systems. Master's thesis, MIT, 2009.
- [Ros07] Kenneth A. Ross. Efficient Hash Probes on Modern Processors. In *ICDE*, 2007.
- [RR99] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, 1999.
- [RR00] Jun Rao and Kenneth A. Ross. Making B⁺-Trees Cache Conscious in Main Memory. In *SIGMOD*, 2000.
- [SGL09] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. k-Ary Search on Modern Processors. In *DaMoN*, 2009.
- [ZR02] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, 2002.

Stets Wertvollständig! – Snapshot Isolation für das Constraint-basierte Datenbank-Caching

Joachim Klein

AG Datenbanken und Informationssysteme
Fachbereich Informatik
Technische Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern
jklein@informatik.uni-kl.de

Abstract: Das Constraint-basierte Datenbank-Caching (CbDBC) erlaubt es, feingranular und dynamisch, Satzmengen häufig verwendeter Prädikate, in der Nähe von Anwendungen vorzuhalten, um lesende Anfragen zu beschleunigen. Dabei lässt sich die Vollständigkeit bzgl. der Anfrageprädikate anhand einfacher Bedingungen (den Constraints) ableiten, welche alle auf dem zentralen Konzept der Wertvollständigkeit aufbauen. Durch das Kopieren der Daten auf den Cache entstehen Replikate, deren Konsistenz zu gewährleisten ist. Gleichzeitig muss jedoch auch die Wertvollständigkeit der Constraints jederzeit gewahrt sein. Wie lässt sich unter diesen Bedingungen für Transaktionen eine akzeptable Isolationsstufe erreichen, wenn der Zugriff auf die primäre Datenbank aufgrund der hohen Latenz teuer und daher zu vermeiden ist? Dieser Aufsatz zeigt, wie sich die Vollständigkeit ganzer Satzmengen im CbDBC wahren lässt, ohne den durch das Caching erreichten Vorteil aufzugeben. Dabei garantiert die für das CbDBC angepasste Synchronisation die Isolationsstufe *Snapshot Isolation* und erlaubt eine verzögerte (lazy) Aktualisierung der Replikate.

1 Motivation

Datenbank-Caching beschleunigt den lesenden Zugriff auf entfernte Datenbanken, indem es Satzmengen, die zur Beantwortung häufig gestellter Anfragen benötigt werden, in der Nähe der betreffenden Anwendungen zur Verfügung stellt. Für alle Datenbank-Caching-Verfahren [LGZ04, The02, ABK⁺03, APTP03, BDD⁺98, LAK10] ist dabei festzulegen, wie die Vollständigkeit der benötigten Sätze bereits im Cache (durch einen *lokalen* Zugriff) bestimmt werden kann. Constraint-basierte Datenbank-Caching (CbDBC) verwendet dazu einfache Bedingungen (*Constraints*), welche die Vollständigkeit von Sätzen bezüglich eines Wertes garantieren. Dies erlaubt es, die einer Anfrage vorgeschaltete Vollständigkeitsprüfung (*Probing* [HB07]) durch die Abfrage einzelner Werte zu realisieren, ohne die benötigten Sätze komplett zu lesen. Gleichzeitig bietet diese Vorgehensweise die Möglichkeit, feingranular und dynamisch zu entscheiden, von welchen Cache-Inhalten eine hohe Lokalität zu erwarten ist.

Eine große Herausforderung beim Datenbank-Caching ist die Realisierung von Synchron-

nisationsverfahren, die für Transaktionen eine hohe Isolationsstufe garantieren. Aufgrund der hohen Latenz zwischen dem zentralen Datenbestand (*Backend*) und dem Cache wurden in der Vergangenheit Ansätze vorgeschlagen, die es erlauben, die Konsistenz stark abzuschwächen, um eine größere Skalierbarkeit zu erreichen [GLRG04a, GLRG04b]. Diese Ansätze haben jedoch zwei entscheidende Nachteile: Einerseits ist es oft nötig, die gewünschte Aktualität der Daten in Anfragen zu spezifizieren, was die Unabhängigkeit der Anwendung vom Datenbanksystem stark beeinträchtigt, und andererseits wird dabei keine wohldefinierte Isolationsstufe erreicht.

Ebenso wie das CbDBC-Verfahren halten auch partielle Replikationsverfahren nur einen Teil der Gesamtdatenmenge als Replikat vor. Die dabei verwendeten Synchronisationsmechanismen bilden deshalb zwar einen wichtigen Startpunkt für die Suche nach einem geeigneten Verfahren, aber die beim CbDBC vorherrschende Dynamik erschwert zumindest die Implementierung solcher Lösungen (vgl. Abschnitt 4).

Durch das redundante Vorhalten der Daten im Cache entstehen (dynamisch veränderliche) Replikate, weshalb die Kontrolle der verteilten Replikate zwingend in den Synchronisationsmechanismus integriert werden muss. Abschnitt 3 beantwortet deshalb kurz die grundlegenden Fragen zur Replikatskontrolle. Eine genaue Diskussion hierzu findet sich in [Kle10].

Zentral für die Auswahl einer geeigneten Nebenläufigkeitskontrolle ist die Forderung, dass der durch das Caching erreichte Geschwindigkeitsvorteil nicht wieder verloren gehen darf. Daher muss ein Verfahren gewählt werden, welches sowohl lesende Zugriffe als auch den Transaktionsabschluss möglichst nicht behindert. Wir konzentrieren uns beim CbDBC auf Verfahren, Snapshot Isolation für Transaktionen garantieren. In Abschnitt 4 erläutern wir diese Entscheidung und erklären die benötigten Grundlagen der Snapshot Isolation in Abschnitt 4.1.

Nachfolgend zeigen wir in Abschnitt 5, wie sich Snapshot Isolation im CbDBC garantieren lässt. Dabei stützen wir uns auf die zentrale Forderung aus [Kle10], auf ältere Snapshots im Backend zugreifen zu können. Der gezeigte Synchronisationsmechanismus erlaubt eine verzögerte (*lazy*) Aktualisierung der Replikate und kommt (abgesehen davon, dass die Änderungen einer Transaktion zum Cache zu übertragen sind) ohne zusätzliche Kommunikation aus. Insbesondere wird kein 2-Phasen-Commit-Protokoll (2PC) zum Abschluss einer Transaktion benötigt.

Bevor wir dies jedoch genau betrachten, wiederholen wir zunächst die zum Verständnis nötigen Grundlagen des CbDBC.

2 Grundlagen des CbDBC

Das Constraint-basierte Datenbank-Caching (CbDBC) beschleunigt Anfragen, indem es Satzmengen häufig angefragter Prädikate in Anwendungsnähe vorhält. Dabei werden die Sätze in sogenannten *Cache-Tabellen* gespeichert und stammen aus einer primären Datenbank, dem Backend. Zu jeder Cache-Tabelle T gehört immer genau eine Backend-Tabelle T_B . Dabei entspricht die Definition einer Cache-Tabelle der ihr zugeordneten Backend-Tabelle bis auf die Fremdschlüsseldefinitionen, die nicht übernommen werden.

Damit der Cache zur Beantwortung eines Prädikats P einer Anfrage Q benutzt werden kann, muss die *Prädikatsextension* von P (dies umfasst alle Sätze, die zur Auswertung von P benötigt werden) im Cache vollständig vorhanden sein. Ist dies der Fall, so ist der Cache hinsichtlich P *prädikatsvollständig* und kann die Beantwortung von Q übernehmen.

Um die Prädikatsvollständigkeit effizient überprüfen und herstellen zu können, benötigt das CbDBC nur zwei Constraint-Typen: Den *referenziellen Cache-Constraint (RCC)* und die sogenannte *Füllspalte (filling column, FC)*. Beide bauen auf dem Konzept der *Wertvollständigkeit* auf, welche die Grundlage des Constraint-basierten Ansatzes bildet.

Definition 2.1 (Wertvollständigkeit) *Ein Wert w einer Spalte $T.a$ ist genau dann wertvollständig (oder kurz vollständig), wenn alle Sätze $\sigma_{a=w}T_B$ im Cache verfügbar sind.*

Ein RCC $S.a \rightarrow T.b$ kann zwischen zwei Spalten mit gleichem Wertebereich definiert werden. Er garantiert jederzeit die Wertvollständigkeit in $T.b$ für alle Werte w aus der *Quellspalte* $S.a$. Hierbei ist zu beachten, dass die Wertvollständigkeit nur für $T.b$, die sogenannte *Zielspalte*, garantiert wird. Dies erlaubt z. B. die Auswertung des Gleichverbundes $S \bowtie_{a=b} T$, falls die Wertvollständigkeit für einen Wert w in S (z. B. für w in $S.c$) gegeben ist, sodass die Anfrage $\sigma_{S.c=w}(S \bowtie_{a=b} T)$ das korrekte Ergebnis liefert. Im umgekehrten Fall (Wertvollständigkeit ist gegeben für einen Wert w aus T) ist die Auswertung des Verbunds nicht möglich¹.

Definition 2.2 (Referenzieller Cache-Constraint, RCC) *Ein RCC $S.a \rightarrow T.b$ von einer Quellspalte $S.a$ und zu einer Zielspalte $T.b$ ist genau dann erfüllt, wenn alle Werte w aus $S.a$ wertvollständig in $T.b$ sind.*

Jede Spalte einer Cache-Tabelle kann auch als Füllspalte ausgezeichnet werden. Mit Hilfe von Füllspalten wird festgelegt, wann und welche Werte vollständig in den Cache geladen werden. Hierzu verwaltet der Cache zu jeder Füllspalte (z. B. $S.f$) eine Menge von *Kandidatenwerten*, für die ein Ladevorgang ausgelöst werden darf.

Definition 2.3 (Füllspalte, FC) *Eine Füllspalte $S.f$ lädt einen Wert w wertvollständig, sobald w von einer Anfrage durch das Prädikat $S.f = w$ explizit referenziert wird und w ein Kandidatenwert ist.*

2.1 Cache Groups

Ein Constraint-basierter Datenbank-Cache verwaltet die für ihn definierten Cache-Tabellen und Constraints (RCCs und FCs) in einer sogenannten *Cache Group*. Eine Cache Group besteht dabei aus einer *Wurzeltabelle (root table)*, in der genau eine Spalte als Füllspalte deklariert ist und evtl. mehreren *Mitgliedstabellen (member tables)*, die über RCCs von der Wurzeltabelle abhängen. Ein einfaches Beispiel hierfür zeigt Abb. 1a, wobei S als Wurzeltabelle mit der Füllspalte $S.f$ darstellt und T bzw. R die Mitgliedstabellen über RCCs von S aus erreichbar sind. Um mehrere Prädikate gleichzeitig zu unterstützen, werden üblicherweise mehrere Cache Groups zu einer *Cache-Group-Föderation* zusammengefasst (vgl. [HB07]).

¹Aus diesem Grund darf ein RCC nicht mit einem Fremdschlüssel gleichgesetzt werden.

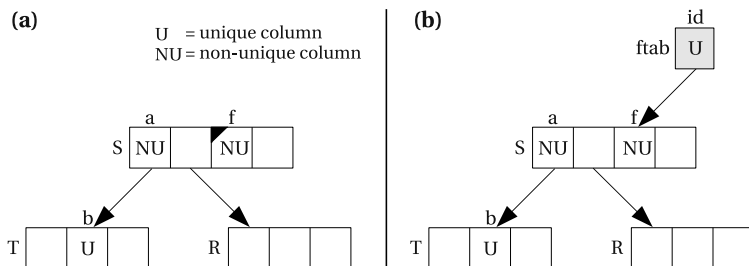


Abbildung 1: Konzeptionelle (a) und interne (b) Darstellung einer Cache Group

Um das Verhalten einer Füllspalte im Cache zu realisieren, wird für jede Füllspalte eine *interne Tabelle* angelegt. Im Gegensatz zu einer Cache-Tabelle hat eine interne Tabelle keine Beziehung zu einer Backend-Tabelle. Für die Füllspalte $S.f$ wird (wie in Abb. 1b gezeigt) eine interne Tabelle $ftab$ angelegt, deren Primärschlüssel, die Spalte $ftab.id$, über einen RCC mit der Füllspalte $S.f$ verbunden wird. Wird nun ein Ladevorgang für den Wert w der Füllspalte $S.f$ ausgelöst, so genügt es den Wert w in Spalte $ftab.id$ einzufügen. Der ausgehende RCC $ftab.id \rightarrow S.f$ erzwingt nachfolgend die erforderliche Wertvollständigkeit von w in $S.f$. Mit Hilfe dieser Vereinfachung kann die Cache-Verwaltung ausschließlich mithilfe von Tabellen und RCCs beschrieben werden.

2.2 Hülle eines Kontrollwertes

Wie zuvor für den Wert w , der durch Einfügung in $ftab.id$ die Wertvollständigkeit in $S.f$ erzwang, gezeigt wurde, kontrolliert jeder *RCC-Quellspaltenwert* die Wertvollständigkeit in den Zielspalten ausgehender RCCs. Aus diesem Grund nennen wir alle Werte, die in RCC-Quellspalten auftreten, *Kontrollwerte*.

Verfolgen wir das Beispiel bzgl. der Cache Group aus Abb. 1 weiter, so fällt auf, dass abhängig von w Sätze im Cache benötigt werden, die ihrerseits wieder Kontrollwerte (z. B. x, y, z in die Spalte $S.a$) einfügen. Die Werte x, y, z müssen dann wiederum in $T.b$ wertvollständig sein. Es entsteht eine rekursive Abhängigkeit von Sätzen unter Verfolgung der ausgehenden RCCs. Daher definieren wir die Menge aller Sätze, die abhängig von einem Kontrollwert w im Cache benötigt werden, als *Hülle des Kontrollwertes* und schreiben für die Hülle dieses Kontrollwertes $C(w)$.

Definition 2.4 (Hülle eines Kontrollwertes) Sei w ein Kontrollwert des RCC $S.a \rightarrow T.b$ und daher $I = \sigma_{a=w} T_B$ die Menge aller Sätze, die in T wertvollständig vorliegen müssen. Die Hülle von w ist rekursiv definiert als Menge aller Sätze $C(w) = I \cup C(w_i), \forall w_i \in W(I)^2$, wobei $W(I) = (w_1, \dots, w_n)$ die Menge aller Kontrollwerte aus I beschreibt.

²Die Menge der Kontrollwerte innerhalb einer Hülle ist endlich und lässt sich (analog zur der Bearbeitung von rekursiven SQL Anfragen) leicht und effizient ermitteln.

3 Replikatskontrolle beim CbDBC

Durch das Kopieren von Sätzen in den Cache entstehen verteilte Replikate, deren Konsistenz zu gewährleisten ist. Dabei spielt die Frage, wo, wie und, vor allem, wann Replikate aktualisiert werden, für die Umsetzung der Nebenläufigkeitskontrolle eine wichtige Rolle. Darüber hinaus muss geklärt werden, auf welchen Replikaten Änderungen durchgeführt werden dürfen. Im Weg weisenden Aufsatz von Gray u. a. [GHOS96] werden die Verfahren daher auch im wesentlichen mittels zweier Parameter kategorisiert.

Der erste Parameter beschreibt, wo Änderungen ausgeführt werden dürfen, auf einer ausgezeichneten Kopie (*primary copy*) oder auf jeder Kopie (*everywhere*). Für das CbDBC wird ein Primary-Copy-Verfahren verwendet [Kle10], da das Backend in natürlicher Weise eine Primärkopie bereitstellt. Die Hauptproblematik besteht jedoch darin, dass die Zulässigkeit einer Update-Ausführung nicht allein durch eine Vollständigkeitsprüfung gewährleistet werden kann. Durch eine Änderung könnten z. B. Trigger ausgelöst oder Integritätsbedingungen verletzt werden. Die zugehörigen Metadaten müssten somit auch auf den Cache kopiert und synchronisiert werden. Darüber hinaus kann niemals sichergestellt werden, dass ein Cache alle Update-Anweisungen einer Transaktion bearbeiten kann. Zum Abschluss einer Transaktion lägen somit an zwei verschiedenen Stellen, im Cache und im Backend, Änderungsinformationen vor, die an andere Caches weiterzuleiten sind. Aus diesen Gründen sind Update-Everywhere-Verfahren für das CbDBC deutlich schwerer umzusetzen. Nach den Ergebnissen aus [GHOS96] benötigt man für Update-Everywhere-Verfahren entweder eine komplexe Nebenläufigkeitskontrolle oder Verfahren zur Konfliktauflösung, welche im CbDBC nicht automatisch, d. h. ohne Benutzerinteraktion, gewährleistet werden kann. Daher werden im CbDBC alle Update-Anweisungen an das Backend weitergeleitet, wo deren Ausführbarkeit stets überprüft werden kann.

Der zweite von Gray u. a. [GHOS96] verwendete Parameter definiert, wann die Replikate aktualisiert werden. Dies kann *direkt* (*eager*) oder *verzögert* (*lazy*) erfolgen. Direkt bedeutet hierbei, dass die Replikate vor Transaktionsabschluss (*Commit*) zu aktualisieren sind, wohingegen die verzögerte Aktualisierung irgendwann nach dem Commit (meist aber möglichst zeitnah) erfolgt.

Eager-Ansätze leiden generell darunter, dass sie den Transaktionsabschluss behindern, bis alle Replikate aktualisiert sind. Im CbDBC wird dieses Problem noch verstärkt, da zwischen dem Backend und den Caches eine hohe Latenz angenommen wird³. Es bietet sich daher an, eine verzögerte Aktualisierungsstrategie im CbDBC anzustreben. Entsprechende Ansätze leiden oft darunter, dass sie Inkonsistenzen erzeugen. Die höchste Isolationsstufe *Serialisierbarkeit* ist dabei meist nicht zu erreichen. Die aktuelle Entwicklung in Forschung und Praxis zielt heute jedoch meist auf eine schwächere Isolationsstufe, die der Snapshot Isolation [BBG⁺95, LKPMJP05, WK05]. In [Kle10] wurden die beiden Umsetzungsvarianten und die dabei auftretenden Herausforderungen für das CbDBC ausführlich diskutiert. Dabei wurde deutlich, dass sich Snapshot Isolation trotz einer verzögerten Aktualisierungsstrategie erreicht lässt, falls die Caches auf den Zustand (Snapshot) im Backend zugreifen können, der ihrem derzeit lokalen entspricht.

³Diese ist oft erst der Grund für den Einsatz eines Datenbank-Cache.

T_1 führt folgende Änderungen durch:

- 1) `UPDATE Order_Lines`
`SET IId = 22`
`WHERE OId = 1;`
- 2) `INSERT INTO Order_Lines`
`VALUES (47, 1, 4);`

In beiden Fällen wird durch die Übernahme der Änderung auf dem Cache der RCC $Order_Lines.IId \rightarrow Items.Id$ verletzt.

Die Hüllen der Kontrollwerte 47 und 22 müssen zunächst nachgeladen werden.

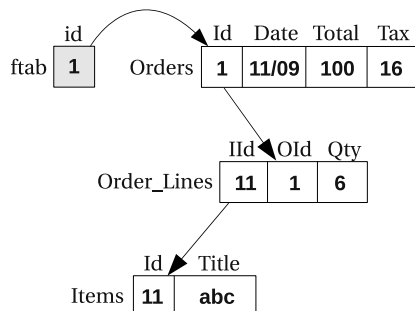


Abbildung 2: Übernahme von Änderungen erfordert das Nachladen von Sätzen

In diesem Aufsatz stützen wir uns auf die vorangestellten Erkenntnisse und verwenden für den in Abschnitt 5 entwickelten Synchronisationsmechanismus eine Replikatskontrolle mit verzögerter Aktualisierung der Replikate unter Verwendung einer Primärkopie.

3.1 Änderungspropagierung

Nachdem wir die Fragen, wo und wann Replikate aktualisiert werden, erörtert haben, betrachten wir in diesem Abschnitt die Besonderheiten bei der Übernahme von Änderungen im Cache und somit die Frage, wie die Replikate (Caches) aktualisiert werden. Wie die Erfassung von Änderungen durch eine geeignete Change-Data-Capture-Strategie erfolgt und durch welche Verfahren die Änderungen selektiv an die Caches gelangen, wird in diesem Aufsatz nicht betrachtet. Erklärungen hierzu finden Sie in [Kle10] und werden zum Verständnis der in diesem Aufsatz entwickelten Nebenläufigkeitskontrolle nicht benötigt.

Eine besondere Bedeutung hat jedoch die Übernahme von Änderungen im Cache. Hierbei sind die dort definierten Constraints jederzeit einzuhalten, wodurch Sätze während der Übernahme nachgeladen werden müssen (vgl. auch [Kle10]). Zur Verdeutlichung dieses Umstandes betrachten wir Abb. 2.

Wir nehmen an, dass die Transaktion T_1 die in Abb. 2 aufgeführten Änderungen im Backend vorgenommen hat. Das ausgeführte Insert-Statement sowie das Update-Statement beziehen sich auf Sätze, die aufgrund des Kontrollwertes $w = Orders.id = 1$ im Cache benötigt werden. Sie gehören zur Hülle von w . Aus diesem Grund müssen die Sätze (47, 1, 4) sowie (22, 1, 6) (nach Änderung) in den Cache eingelagert werden. Damit der RCC $Order_Lines.IId \rightarrow Items.Id$ nicht durch die Übernahme der Änderungen verletzt wird, müssen zunächst die Hüllen der Kontrollwerte $u = 22$ und $v = 47$ aus $Order_Lines.IId$ nachgeladen werden.

Durch Änderungen können Sätze auch ihre Abhängigkeiten verlieren. Im Beispiel könnte

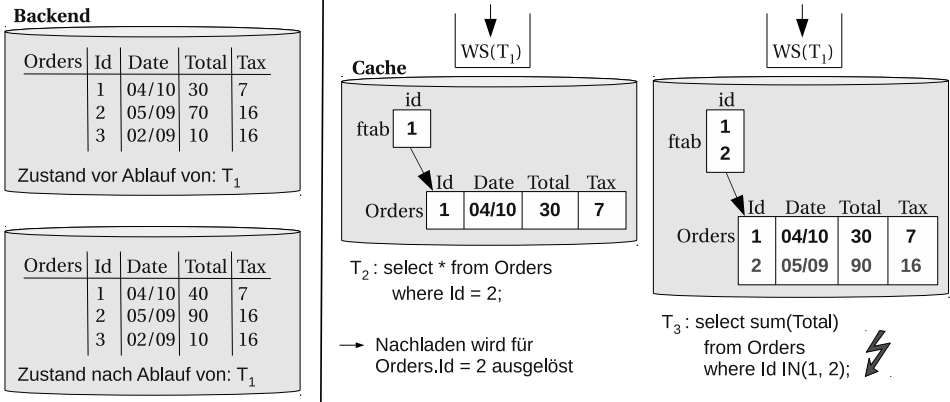


Abbildung 3: Nachladen verursacht ein Einlagern verschiedener transaktionskonsistenter Zustände

der Satz (11, *abc*) in *Items* z. B. entladen werden. Durch Wegfall von Abhängigkeiten werden jedoch niemals RCCs verletzt und deshalb ist hierbei auch kein Nachladen nötig. Ein *Garbage Collector* sucht und löscht nicht mehr benötigte Sätze nebenläufig.

Die Tatsache, dass bei der Übernahme von Änderungen (und auch durch Füllspalten) Ladeprozesse ausgelöst werden, erschwert die Gewährleistung der Konsistenz im Cache nachhaltig. Wir wollen dies durch ein weiteres Beispiel (vgl. Abb. 3) verdeutlichen, wobei wir annehmen, dass die Caches verzögert aktualisiert werden und im Backend nur die neueste Version von Sätzen gelesen werden kann.

Im Beispiel nehmen wir an, der Cache hat die Bestellung 1 (*Orders.Id* = 1) vor dem Ablauf der Transaktion T_1 geladen. Nachfolgend wird im Backend die Änderungstransaktion T_1 ausgeführt, welche die Summe (*Orders.Total*) für die Bestellung 1 und 2 auf die Werte 40 bzw. 90 ändert. Die Änderungen von T_1 wurden bereits in Form eines *Write Set* (WS) an den Cache übertragen, der aber die Änderungen zu diesem Zeitpunkt noch nicht übernommen hat. In der Zwischenzeit wird über den Cache mit der Ausführung der Transaktion T_2 begonnen, welche das Nachladen der Bestellung 2 auslöst, wobei nun im Backend nur noch der Zustand nach Ablauf von T_1 zugreifbar ist. Auf dem Cache befinden sich nun Sätze, die den Zustand vor Ablauf von T_1 repräsentieren, und Sätze, die den Zustand nach Ablauf von T_1 aufweisen, wodurch die Ausführung von T_3 ein *Incorrect Summary* erzeugt.

Die aufgeführten Beispiele machen deutlich, dass im Cache nur Sätze des gleichen transaktionskonsistenten Zustandes eingelagert sein dürfen, damit dieser überhaupt verwendet werden kann. Dies ist nur dann möglich, wenn der Cache während des Nachladens noch Zugriff auf den Zustand hat, den er selbst gerade repräsentiert.

Basierend auf den bisher gewonnenen Erkenntnissen wollen wir nachfolgend untersuchen, welche Verfahren zur Nebenläufigkeitskontrolle im CbDBC einsetzbar sind. Dabei werden wir zeigen, dass sich die erwünschten Eigenschaften und die notwendige Konsistenz nur durch den Einsatz von Mehrversionenverfahren garantieren lassen, wobei die gewählte

Isolationsstufe der Snapshot Isolation lesende Zugriffe nicht behindert.

4 Nebenläufigkeitskontrolle beim CbDBC

Die gewählte Art der Replikatskontrolle muss mit einer geeigneten Nebenläufigkeitskontrolle integriert werden [LKPMJP05, WK05]. In diesem Aufsatz bewerten wir dabei die verschiedenen Verfahren zur Nebenläufigkeitskontrolle nur im Bezug auf die von uns gewünschte verzögerte Änderungsübernahme und unter dem Aspekt, dass eine Primärkopie existiert. Das wichtigste Kriterium für die Auswahl eines geeigneten Verfahrens für das CbDBC ist, dass der durch das Caching erzielte Vorteil (die Performance, die durch eine lokale Anfragebeantwortung gewonnen wurde) nicht verloren gehen darf. Daher sind Verfahren, die einen Zugriff aufs Backend benötigen, z. B. um eine Sperre anzufordern wie beim verteilten 2-Phasen-Sperrprotokoll (D2PL [BG81]), nicht geeignet. Beim Einsatz von optimistischen Verfahren, wie z. B. *Forward Oriented Concurrency Control (FOCC)* oder *Backward Oriented Concurrency Control (BOCC)*, müsste das *Read Set* der Transaktion im Cache erfasst und zur Verifikation an das Backend übermittelt werden.

Als zweites zentrales Problem muss der Cache, um die Wertvollständigkeit seiner Constraints zu erhalten, Sätze nachladen. Dazu ist es notwendig, auf einen bestimmten transaktionskonsistenten Zustand im Backend zugreifen zu können, wie bereits im vorangehenden Abschnitt gezeigt. Um dies effizient gewährleisten zu können, muss auf Cache und Backend eine Mehrversionenverfahren zum Einsatz kommen.

Der Einsatz von Mehrversionenverfahren ist auch dadurch motiviert, dass für eine Transaktion zu jeder Zeit gewährleistet sein muss, dass sie den gleichen transaktionskonsistenten Zustand liest, egal ob sie auf den Cache oder das Backend zugreift. Da sich der Cache-Inhalt fortlaufend ändert, kann nicht gewährleistet werden, dass das Lesen einer Transaktion nur auf den Cache beschränkt ist. Durch die verzögerte Aktualisierungsstrategie unterscheiden sich die neuesten transaktionskonsistenten Zustände der Datenbanken voneinander. Somit müssen ältere transaktionskonsistente Zustände im Backend aufgehoben werden, damit sie für den Cache zugreifbar sind, wodurch zwingend der Einsatz eines Mehrversionenverfahrens notwendig wird.

Zur effizienten Implementierung von Snapshot Isolation [Fek09] werden auch Mehrversionenverfahren eingesetzt. Als wesentliche Eigenschaft blockiert Snapshot Isolation niemals lesende Zugriffe. Gelesene Objekte müssen somit auch nicht aufgezeichnet werden, was für CbDBC von großem Vorteil ist. So wird die Implementierung Middleware-basierter Lösungen sehr erleichtert, da es oft keine Unterstützung gibt, alle gelesenen Objekte einer Transaktion abzufragen bzw. in einem Read Set zu speichern. Durch die Einführung und Einhaltung zusätzlicher Bedingungen lässt sich für SI-Transaktionen sogar Serialisierbarkeit garantieren (vgl. [CRF08, FLO⁺05]). Um dies jedoch innerhalb des Synchronisationsmechanismus (also nicht durch die Applikationslogik) zu gewährleisten müssen Leseoperationen jedoch wiederum beachtet werden.

Im Nachfolgenden betrachten wir Snapshot Isolation und später deren Umsetzung im CbDBC genauer. Dabei gehen wir stets davon aus, dass auf eine Objektmenge oder ein

Objekt O abstrakt zugegriffen wird. Für die Nebenläufigkeitskontrolle spielt es nämlich keine Rolle, ob O einzelne Attributwerte, Sätze oder Seiten darstellt. Wichtig ist nur, das über die Art der Versionskontrolle immer der gleiche transaktionskonsistente Zustand zugegriffen wird.

4.1 Snapshot Isolation

Die Isolationsstufe Snapshot Isolation wurde maßgeblich in [BBG⁺95] motiviert. Snapshot Isolation gründet direkt auf dem Einsatz von Mehrversionenverfahren, die einer Transaktion stets den Zustand (Snapshot), der zu ihrem Startzeitpunkt (*Begin of Transaction*, *BOT*) gerade gültig war, präsentieren. Dabei muss dieser Zeitpunkt irgendwann vor dem ersten Lesezugriff durch die Transaktion festgelegt werden. Lesende Zugriffe werden niemals blockiert. Sie sind also immer möglich, was für das Datenbank-Caching von besonderer Bedeutung ist. Will eine Transaktion T_x ein Objekt O schreiben, so darf sie dies im einfachsten Fall nur dann, wenn nach der zu lesenden Version V_i keine weitere festgeschriebene Version $V_{j>i}$ existiert. Ist dies der Fall, wird T_x zurückgesetzt. Dieses Vorgehen bezeichnet man auch als *First Committer Wins* [BBG⁺95]. Es werden also nur sogenannte Schreib/Schreib-Konflikte erkannt und aufgelöst. Schreibende Transaktionen müssen also nicht durch ein spezielles Protokoll (im einfachsten Fall ein RX-Sperrprotokoll) synchronisiert werden.

Um den BOT einer Transaktion bzw. den Commit-Zeitpunkt einer Version festlegen zu können, benötigt man eine logische Uhr, die zumindest nach dem erfolgreichen Commit einer schreibenden Transaktion erhöht werden muss. Um auch den BOT einer Transaktion eindeutig zuordnen zu können, wird diese Uhr, welche wir nachfolgend als *Systemuhr* t bezeichnen, meist auch bei BOT um eins erhöht.

In unserer Implementierung gehen wir entsprechend vor: Bei Beginn einer Transaktion wird der Wert von t als BOT festgehalten; danach wird t um eins erhöht. Gleiches gilt beim erfolgreichen Abschluss einer Transaktion (z. B. von T_x), wobei alle von ihr geschriebenen Versionen mit dem Zeitpunkt $EOT(T_x) = t$ markiert werden.

Bei der Umsetzung von Snapshot Isolation für verteilte Datenbanksysteme unterscheidet man üblicherweise zwischen *starker* (*strong*) und *schwacher* (*weak*) Snapshot Isolation [DS06]. Der Unterschied besteht darin, dass es im verteilten Fall dazu kommen kann, dass eine Transaktion ihre eigenen Änderungen nicht sofort sieht, je nachdem wann Replikate aktualisiert werden. Bei starker Snapshot Isolation kann dies nicht vorkommen, bei schwacher Snapshot Isolation wird dies toleriert.

Die nachfolgend beschriebene Vorgehensweise beim CbDBC garantiert zunächst nur schwache Snapshot Isolation. In Abschnitt 5.4 erklären wir aber auch, wie sich starke Snapshot Isolation garantieren lässt.

5 Snapshot Isolation für CbDBC

Wir betrachten nun, wie sich Snapshot Isolation beim CbDBC erreichen lässt. Da das Backend und die Caches auf unterschiedlichen Hosts agieren, unterscheiden wir die *Backend-seitige Systemuhr* t_{BE} von der *Cache-seitigen Systemuhr* t_{C_i} . Wir benötigen diese Unterscheidung auch, um deutlich zu machen, dass alle Instanzen ihre eigene (lokal verwaltete) Versionskontrolle besitzen. Eine global eindeutige Systemuhr wird somit nicht benötigt.

In den nachfolgenden Betrachtungen beziehen sich alle Erklärungen direkt auf die Versionen, die bei Änderung eines Objektes angelegt werden. Dabei spielt es jedoch keine Rolle, ob die vorgeschlagene Versionskontrolle direkt in ein eigenständiges (evtl. proprietäres) CbDBC-System integriert wird oder ob das Caching-System Middleware-basiert umgesetzt ist, wobei jedoch nur Datenbanksysteme benutzt werden können, die lokal auch Snapshot Isolation anbieten, wie z. B. PostgreSQL [Pos10], Oracle [Ora10] oder MS-SQL-Server [Mic10] (vgl. hierzu auch Abschnitt 5.7).

Im Folgenden bezeichnen wir eine durch den Benutzer (bzw. zugreifende Applikation) initialisierte Transaktion T_x als *global* bzw. als *Benutzertransaktion*. Jede globale Transaktion wird immer genau von einem Cache C_i zusammen mit dem Backend ausgeführt, wobei das *Cache-Management-System* (*CacheMS*) die Transaktion T_x kontrolliert. Für den Teil der Transaktion T_x , der über das Backend ausgeführt wird, schreiben wir T_x^{be} und für den Cache-Zugriff $T_x^{c_i}$. Wir bezeichnen diese Teile von T_x als *lokale* Transaktionen, da sie nur einen lokalen Zugriff, entweder auf die Cache-Datenbank oder auf die Backend-Datenbank zulassen. Bei einer Middleware-basierten Umsetzung sind für die Zugriffswege T_x^{be} und $T_x^{c_i}$ eigenständige, sogenannte *echte* lokale Teiltransaktionen nötig, die vom CacheMS initialisiert werden. Darüber hinaus gibt es weitere lokale Transaktionen, die durch das CacheMS initialisiert werden, um bestimmte Verwaltungsaufgaben (z. B. das Nachladen) von Werten durchzuführen. Diese Transaktionen werden einfach mit ihrem Zweck markiert. Wir schreiben T_y^{load} für eine lokale Ladetransaktion, die auf das Backend zugreift, um die zu landenden Sätze zu lesen, und T_y^{insert} für deren Einfügeoperationen im Cache. Lokale Transaktionen, die die Übernahme von Änderungen aus dem WS realisieren, markieren wir mit *accept*, so ergibt sich z. B. T_z^{accept} . Lokale Transaktionen, die auf das Backend zugreifen, nennen wir auch einfach *Backend-Transaktionen* und dementsprechend lokale Transaktionen, die auf den Cache verweisen *Cache-Transaktionen*. Auch wenn die Ausführung lokaler Transaktionen nicht Middleware-basiert, sondern integriert (intern) abläuft, gehen wir davon aus, dass für diese Transaktionen die ACID-Eigenschaften unter Isolationsstufe Snapshot Isolation eingehalten werden. Den zugewiesenen Zugriffszeitpunkt von Backend-Transaktionen benennen wir t_Z^{be} und von Cache-Transaktionen $t_Z^{c_i}$.

5.1 Referenzierung eines Snapshots

Wie bereits erwähnt, muss der Cache ältere Versionen (Snapshots) im Backend referenzieren und zugreifen können. Dies wird durch die Übermittlung des gewünschten *Zugriffs-*

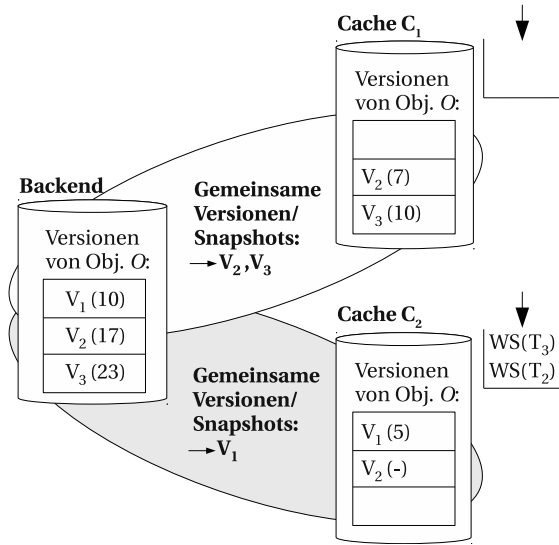


Abbildung 4: Gemeinsame Versionen von Backend und Cache bei verzögerter Aktualisierung

zeitpunktes t_Z^{be} ans Backend ermöglicht. Es genügt beim ersten Lesen einer Transaktion im Backend, diesen Zeitpunkt mitzuteilen. Die Frage ist jedoch, woher der Cache diesen Zeitpunkt kennt bzw. wie er ihn herausfindet. Das Backend überträgt den Zustandszeitpunkt t_Z beim Abschluss einer globalen Transaktion T zusammen mit den Änderungen von T im WS. Dabei ist $EOT(T) = t_{BE} = t_Z$. Sobald der Cache alle Änderungen aus diesem WS übernommen hat, stellt er zusammen mit dem Backend den neuen Zustand (Snapshot), der nach T erreicht wurde, bereit und kann daher fortan unter Angabe des im WS übermittelten Zustandszeitpunktes t_Z auf das Backend zugreifen. Die Initialisierung stellt dabei einen Sonderfall dar, weil hierbei zunächst ein Startzeitpunkt vom Backend zugewiesen wird (vgl. Abschnitt 5.2). Wir betrachten hierzu Abb. 4, welche einen Überblick über die angestrebte verteilte Versionsverwaltung bietet.

In Abb. 4 sind im Backend die Versionen V_1 , V_2 und V_3 des Objektes O zu den Zeitpunkten 10, 17 und 23, welche in Klammern angegeben sind, festgeschrieben worden. Wir nehmen an, dass diese Versionen durch die Transaktionen T_1 , T_2 und T_3 erzeugt wurden, die zu den gleichen Zeitpunkten (z. B. $EOT(T_1) = 10$) erfolgreich abgeschlossen wurden. Da auf einem Cache niemals Änderungen durch einen Benutzer stattfinden, werden dort nur Versionen von O erzeugt, wenn O neu geladen wird oder wenn ein WS nachfolgend eine Änderung für O signalisiert.

In unserem Beispiel lädt der Cache C_1 die Version V_2 und schreibt diese zum Zeitpunkt 7 seiner lokalen Systemuhr (t_{C_1}) fest. Das Überspringen von älteren Versionen entsteht, wenn der Cache erst zu einem Zeitpunkt $t_{BE} > 17$ das Objekt O lädt. Dies wäre z. B. der Fall, wenn die WSs von T_1 und T_2 übernommen werden, bevor der Cache O lädt. Im Beispiel hat C_1 auch bereits das WS von T_3 eingespielt und die Änderung auf O zum Zeitpunkt $t_{C_1} = 10$ festgeschrieben.

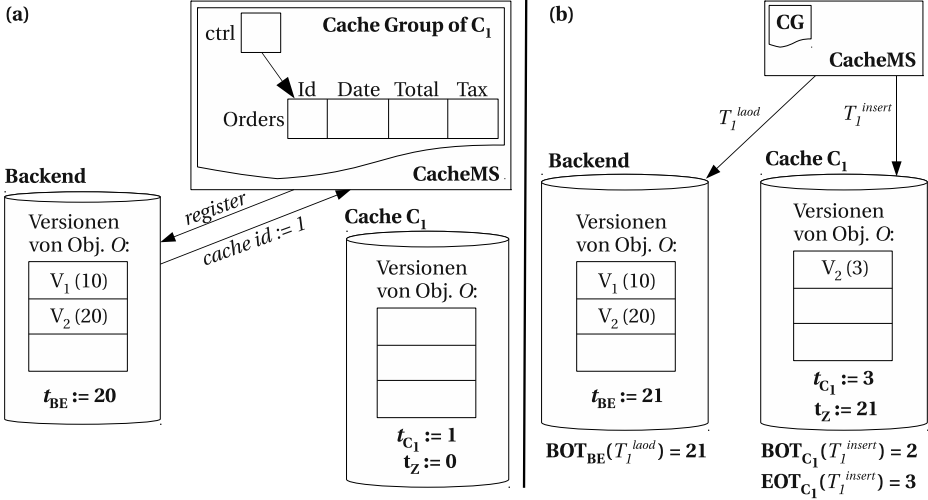


Abbildung 5: Initialisierung des Caches

Nehmen wir an, eine Transaktion T_{neu} startet, bevor das WS von T_3 eingespielt wird. In diesem Fall wird der Snapshot von T_{neu} korrekt referenziert, wenn zum Cache im Zeitintervall $t_{C_1} \geq 7 \wedge t_{C_1} < 10$ zugegriffen wird und zum Backend zum Zeitpunkt t_Z von 17. Die Wahl eines späteren Zeitpunktes als $t_Z^{be} = 17$ würde im Backend möglicherweise einen anderen Snapshot adressieren, wenn nämlich eine Transaktion zum Zeitpunkt 18 Versionen anderer Objekte eingebracht hätte. Im Cache repräsentieren alle Zeitpunkte t_{c_i} zwischen der Übernahme zweier WSs den gleichen transaktionskonsistenten Backend-Zustand. Daher sind hier mehrere Zugriffszeitpunkte $t_Z^{c_i}$ wählbar.

Der Cache C_2 hat bereits die Version V_1 von O geladen und das WS von T_1 übernommen. Die Änderungen der WSs von T_2 und T_3 stehen noch aus. Somit kann C_2 einer Transaktion T_{neu} zurzeit nur einen gemeinsamen Snapshot basierend auf der gemeinsamen Version V_1 anbieten.

5.2 Initialisierung

Ein neu zu initialisierender Cache ist zunächst leer. Damit er seine Arbeit aufnehmen kann, muss er sich zunächst am Backend registrieren (vgl. Abb. 5a). Dabei erhält er eine eindeutige Identifikation, die *Cache-Id*, um sich während der weiteren Verarbeitung gegenüber dem Backend ausweisen zu können. Der Cache ist bereits vor dieser Registrierung in der Lage, Benutzertransaktionen entgegenzunehmen. Diese werden solange komplett an das Backend weitergeleitet, bis die Initialisierung vollständig abgeschlossen ist.

Nachdem die Registrierung abgeschlossen ist, greift der Cache auf das Backend durch Initialisierung einer Ladetransaktion T_1^{load} erstmalig zu. Da der Cache noch leer ist, wird

für T_1^{load} der Wert $t_Z^{be} = t_Z = 0$ als Zugriffsparameter angegeben. Dies signalisiert dem Backend, dass T_1^{load} Zugriff auf den letzten festgeschriebenen Datenbankzustand (den neuesten Snapshot) erhalten soll. Diese Situation ist in Abb. 5b dargestellt, wobei der tatsächliche Zugriffszeitpunkt für T_1^{load} auf $BOT_{BE}(T_1^{load}) = t_{BE} = 21$ korrigiert wird. Dieser erstmals festgelegte Zeitpunkt wird als *Ausgangszeitpunkt* des Caches im Backend hinterlegt. Der Cache ist nun initialisiert und kann seine normale Verarbeitung aufnehmen. Muss der Cache in dieser Phase ($t_Z = 0$) noch weitere lokale Transaktionen anlegen, die auf das Backend zugreifen (z. B. falls eine Benutzertransaktion neu startet), so werden diese auf den Ausgangszeitpunkt (hier 21) festgelegt. Auf diese Weise muss dem Cache niemals durch eine gesonderte Kommunikation der Ausgangszeitpunkt explizit mitgeteilt werden.

Start der Auslieferung von WSs. Sobald der Cache erstmalig eine lokale Transaktion zum Zugriff auf das Backend gestartet hat (T_1^{load}), werden alle WSs von Änderungs-transaktionen mit einem $EOT_{BE} \geq 21$ an den Cache ausgeliefert. Ändern sich also Objekte, die mittels T_1^{load} geladen wurden, kann der Cache alle Folgeänderungen anhand der ihm zugestellten WSs nachvollziehen. Da in jedem WS der nachfolgend zu verwendende Zustandszeitpunkt t_Z mitgeschickt wird, ist dieser nach der ersten WS-Übernahme im Cache eindeutig definiert.

Transaktionskonsistenter Zustand. Wir betrachten nochmals Abb. 5b. Solange C_1 kein WS abschließend eingespielt hat (also auch während der Übernahme eines WS) werden alle geladenen Sätze über T_1^{load} abgefragt. Somit erreichen nur Objekte des gleichen Snapshots (d. h. des gleichen transaktionskonsistenten Zustands) den Cache. Somit spielt es keine Rolle, wann und wie oft Sätze in die Cache-Datenbank eingefügt werden. In unserem Beispiel aus Abb. 5b wurde das Objekt O zum Zeitpunkt $EOT_{C_1}(T_1^{insert}) = 3$ in die Cache-Datenbank eingebracht. Eine lokale Transaktion $T_{alt}^{c_1}$ mit $BOT_{C_1}(T_{alt}^{c_1}) = 1$ kann auf das Objekt O im Cache nicht zugreifen, eine Transaktion $T_{neu}^{c_1}$ mit $BOT_{C_1}(T_{neu}^{c_1}) = 4$ hingegen schon. Die durch das CbDBC vorgegebenen Regeln, die eine korrekte Cache-Verwaltung (bzgl. Laden, Entladen, Probing) garantieren, bleiben dabei unangetastet und müssen natürlich eingehalten werden [KB09, HB07].

5.3 Übergang zum Nachfolgezustand

Sobald ein WS im Cache eintrifft, muss dieses schnellstmöglich verarbeitet werden. Dabei muss sichergestellt werden, dass die WSs in der Commit-Reihenfolge ihrer zugehörigen Änderungstransaktionen abgearbeitet werden. Jede WS-Übernahme wird durch genau eine lokale Transaktion im Cache abgewickelt (z. B. von T_1^{accept} , wie in Abb. 6a gezeigt).

Für jede Änderung im WS ist zu prüfen, ob der Cache den als geändert aufgeführten Satz⁴ überhaupt enthält. Ist der Satz im Cache, wird er durch T_1^{accept} angepasst. Wie in

⁴Im WS werden alle Attribute eines geänderten Satzes für jede Cache-Tabelle in ihrem alten und neuen Zustand übertragen (vgl. [Kle10]).

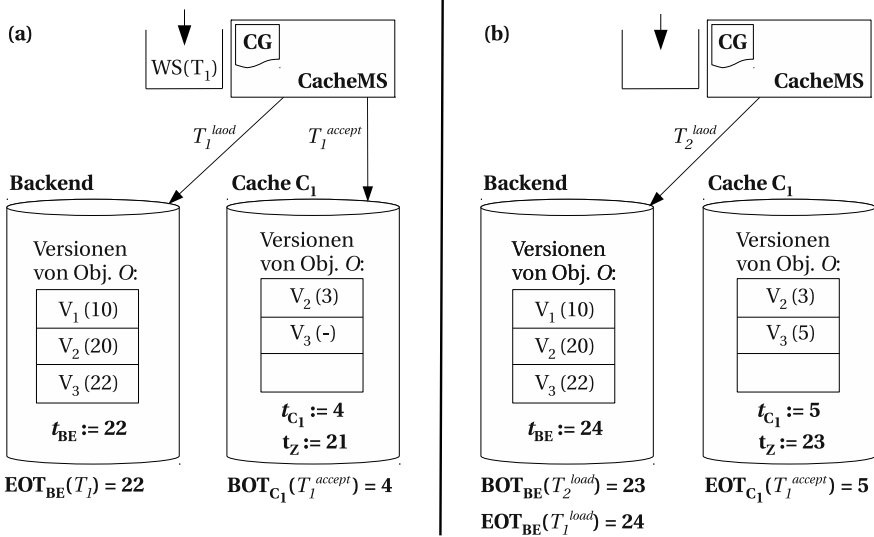


Abbildung 6: Übergang zum nächsten transaktionskonsistenten Zustand.

Abschnitt 3 gezeigt, können hierbei Ladeoperationen ausgelöst werden. Dabei ist es sogar möglich, dass neue Sätze den Cache erreichen, für die im gerade abzuarbeitenden WS eine Änderung aufgeführt ist, welche dann noch zusätzlich (nach dem entsprechenden Ladevorgang) durchzuführen ist. Diese könnte wieder ein Nachladen auslösen usw. Es entsteht eine rekursive Abarbeitung des WS. Dabei wird jedoch jede im WS aufgeführte Änderung höchstens ein Mal angewandt, wodurch der Übernahmeprozess stets terminiert. Erst wenn alle anwendbaren Änderungen des einzuspielenden WS übernommen wurden und die hierdurch ausgelösten Nachladeoperationen beendet sind, wird T_1^{accept} abgeschlossen. Hierdurch wird der Cache atomar von einem transaktionskonsistenten Zustand in den nachfolgenden überführt.

Nach dem erfolgreichen Abschluss von T_1^{accept} müssen alle nachfolgenden Ladeprozesse mit einer lokalen Backend-Transaktion durchgeführt werden, die den neuen transaktionskonsistenten Zustand repräsentiert. In Abb. 6b wurde dazu die Transaktion T_2^{load} neu eingerichtet, welche mit dem Zugriffszeitpunkt $t_Z^{be} = t_Z = 23$ aus dem WS von T_1 initialisiert wurde. Ebenso muss jede neu zu initialisierende lokale Backend-Transaktion T_{neu}^{be} mit $t_Z^{be} = 23$ angelegt werden. Die genaue Initialisierung der zugehörigen Benutzertransaktion T_{neu} wird im anschließenden Abschnitt besprochen.

5.4 Snapshot-isolierte Benutzertransaktion

Durch die in den beiden vorangegangenen Abschnitten beschriebene Versionsverwaltung sind wir nun in der Lage einer Benutzertransaktion (z. B. T_1) einen eindeutigen, globalen

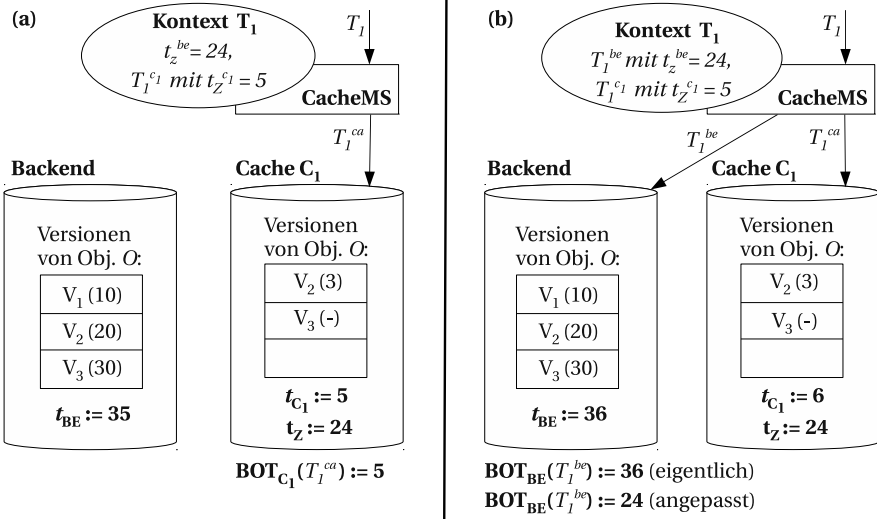


Abbildung 7: Initialisierung einer Benutzertransaktion.

Snapshot zuzuweisen. Dieser ergibt sich durch die Festlegung der Zugriffszeitpunkte für die lokalen Teiltransaktionen T_1^{be} und $T_1^{c_1}$. Sie werden im *Transaktionskontext* (oder kurz: Kontext) von T_1 hinterlegt, der vom CacheMS verwaltet wird. Auf diese Weise wird durch die Informationen im Kontext der Snapshot der Transaktion definiert. Aus diesem Grund bezeichnen wir solche Transaktionskontexte fortan synonym als Snapshot der Transaktion.

Jeder Snapshot wird gebildet, indem zunächst $T_1^{c_1}$ für die Cache-Datenbank angelegt wird. Hierbei wird kein Zugriffszeitpunkt vorgegeben. Der im Snapshot hinterlegte Zeitpunkt für den Cache-Zugriff ergibt sich direkt aus dem Startzeitpunkt von $T_1^{c_1}$. Somit ist $t_z^{c_1} = BOT_{c_1}(T_1^{c_1}) = t_{c_1}$ der im Kontext hinterlegte Cache-Zugriffszeitpunkt. Gleichzeitig wird im Kontext von T_1 der aktuelle Wert von t_z für den lokalen Backend-Zugriff hinterlegt ($t_z^{be} = t_z$). Die Transaktion T_1^{be} wird erst initialisiert, wenn ein Zugriff auf das Backend tatsächlich nötig wird. So wird für Transaktionen, die nur Lesezugriffe ausführen, die durch den Cache beantwortbar sind, das Backend nicht involviert. Wir betrachten das Anlegen einer Benutzertransaktion nochmals genau mit Hilfe der Beispiele aus Abb. 7.

In Abb. 7a greift die Benutzertransaktion T_1 erstmalig über das CacheMS von C_1 auf das CbDBC-System zu. Die lokale Systemuhr des Caches hat derzeit den Wert $t_{c_1} = 5$ und im letzten, bereits eingespielten WS wurde der Zustandszeitpunkt $t_z = 24$ übermittelt. Der Cache initialisiert den Snapshot von T_1 , indem er die Zeitpunkte $t_z^{c_1} = BOT_{c_1}(T_1^{c_1}) = t_{c_1} = 5$ und $t_z^{be} = t_z = 24$ für T_1 festlegt. Danach wird die lokale Systemuhr t_{c_1} um eins erhöht ($t_{c_1} = 6$, vgl. Abb. 7b). Die lokale Backend-Transaktion T_1^{be} ist noch nicht initialisiert. T_1 kann zurzeit also nur die Version V_2 von O im Cache lesen. Im Bild ist auch angedeutet, dass der Cache gerade die Version V_3 übernimmt (z. B. durch T_1^{accept}), die im Backend bereits festgeschrieben wurde. T_1^{accept} ist jedoch noch nicht abgeschlossen und daher zu T_1 nebenläufig. Sie endet erst zu einem Zeitpunkt $t_{c_1} > 5$, sodass T_1 die Version

V_3 niemals lesen kann.

Der Cache hat den Snapshot gebildet und ist nun in der Lage, falls T_1 auf das Backend zugreifen muss, T_1^{be} jederzeit korrekt anzulegen (vgl. Abb. 7b). Der Zugriffszeitpunkt von T_1^{be} würde normalerweise (unangepasste Snapshot Isolation) auf den tatsächlichen Beginn der Transaktion $BOT_{BE}(T_1^{be}) = t_{BE} = 36$ festgelegt. Da der Cache für T_1^{be} aber den Zugriffszeitpunkt 24 hinterlegt hat, wird der Beginn auf $BOT_{BE}(T_1^{be})$ auf den Wert 24 vorverlegt. Egal ob T_1 nun auf den Cache oder das Backend zugreifen muss, er sieht immer die gleiche Version des Objektes O (hier V_2). Da bereits eine weitere Version (V_3) von O durch eine andere Transaktion festgeschrieben wurde, kann T_1 auf O keine Änderung vornehmen. Dies würde zu einem Schreib/Schreib-Konflikt führen, der im Backend wie üblich (nach den Regeln für SI) erkannt wird. T_1 müsste dann zurückgesetzt werden (First Committer Wins).

Wir konnten somit zeigen, wie im Cache ein transaktionskonsistenter Zustand hergestellt und gewahrt werden kann. Durch die Möglichkeit, ältere Versionen im Backend zu referenzieren, kann jeder Benutzertransaktionen ein einheitlich aufgebauter, global konsistenter Snapshot zur Verfügung gestellt werden. Da Änderungen nur im Backend erfolgen, lassen sich dort alle Schreib/Schreib-Konflikte korrekt erkennen. Insbesondere der erfolgreiche Transaktionsabschluss kann allein durch das Backend verifiziert werden. Ein 2PC-Protokoll ist nicht notwendig, da auf den Caches keine Dauerhaftigkeit von Objekten benötigt wird. Tritt während der Verarbeitung ein schwerwiegender Fehler auf (wenn z. B. der Cache ausfällt oder ein übermitteltes WS nicht lesbar ist), kann der Cache im einfachsten Fall geleert und neu initialisiert werden.

Der beschriebene Ansatz garantiert jedoch nur schwache Snapshot Isolation, da es vorkommen kann, dass eine Transaktion ein Objekt ändert (Backend), welches sie nachfolgend nochmal liest (Cache). Wir diskutieren im nachfolgenden Abschnitt kurz einige Möglichkeiten, starke Snapshot Isolation zu erreichen, falls schwache Snapshot Isolation nicht ausreicht.

5.5 Starke Snapshot Isolation

Um starke Snapshot Isolation zu erreichen, muss sichergestellt sein, dass eine Transaktion T_1 keine Sätze im Cache liest, die sie selbst bereits geändert hat. Am einfachsten gelingt dies, wenn einer Transaktion der Zugriff auf den Cache verwehrt bleibt, sobald sie einmal ein Update-Statement ausführt hat. Dadurch wird jedoch die Nutzung des Caches sehr stark eingeschränkt. Reine Lesetransaktionen und Transaktionen, die erst am Ende schreiben, würden jedoch kaum oder gar nicht beeinflusst.

Um diese radikale Lösung zu vermeiden, kann man selektiv nur Zugriffe auf Cache-Tabellen verbieten, deren zugeordnete Backend-Tabelle zuvor von einem Update-Statement geändert wurde. Da jedes Statement einer Transaktion zunächst am Cache ankommt kann dieser die betroffene Tabelle auslesen und das Zugriffsverbot für die Transaktion in deren Kontext vermerken.

Als weitere Verfeinerung können alle Primärschlüssel (oder die *Record Identifier*) geän-

derter Sätze an die Antwortnachricht einer Update-Anweisung angehängt werden (per *Piggybacking*). Dies macht nur dann Sinn, wenn nicht zu viele Informationen zu übertragen sind. Der Cache kann so prüfen, ob zu lesende Sätze bereits durch die Transaktion geändert wurden. Wenn ja, wird die Leseoperation vom Backend beantwortet. Wurden zu viele Sätze geändert, so dass der Verwaltungsaufwand zu hoch ist, werden nachfolgend ungeprüft alle Operationen der Transaktion direkt ans Backend geschickt. Der große Nachteil dieser Vorgehensweise besteht darin, dass die Änderungen einer Anweisung direkt im Backend ermittelt werden müssen (z. B. durch Trigger). Das im Aufsatz gezeigte Verfahren vermeidet dies gerade, da hierbei die Änderungen erst nach Transaktionsabschluss (z. B. durch Log-Sniffing-Techniken) ermittelt werden können.

Eine viel versprechende Methode ist die Durchführung einer Vollständigkeitsprüfung (Probing) für Update-Anweisungen. Ist das Probing erfolgreich, kann der Cache selbst die geänderten Sätze ermitteln und daraus auch die neuen Werte berechnen. So kann der Cache selbst einer Transaktion ihre geänderten Sätze zurückliefern, falls die Ausführung im Backend erfolgreich war. Hierbei ist es jedoch zu beachten, dass dem Backend das erfolgreiche Probing signalisiert wird. Das Backend muss im Gegenzug die Zulässigkeit des Update bestätigen. Werden beim Update Folgeänderungen ausgelöst, die wiederum den momentanen Cache-Inhalt beeinflussen, muss dies zurückgemeldet werden, da der Cache solche Situationen nicht erkennen kann.

Es lassen sich auch Kombinationen der vorgeschlagenen Verfahren implementieren. Sobald der Cache jedenfalls transaktionslokal geänderte Sätze – in der Regel bei wiederholtem Lesen – nicht selbst zur Verfügung stellen kann, muss die Leseoperation im Backend erfolgen.

Alle hier aufgeführten Methoden können starke Snapshot Isolation erreichen. In unserem CbDBC-Prototyp ACCache [BHM06] ist jedoch bisher nur schwache Snapshot Isolation umgesetzt.

5.6 Löschung alter Versionen

Alte Versionen von Objekten dürfen gelöscht werden, wenn keine Transaktion mehr auf sie zugreifen kann. Sobald die Caches ein WS übernehmen, erhöht sich ihr lokaler Zustandszeitpunkt t_Z . Legt der Cache eine Backend-Transaktion an, wird der Zugriffszeitpunkt t_Z^{be} ans Backend zurück übermittelt. t_Z und somit auch t_Z^{be} werden stets nur erhöht. Eine vom Cache angelegte Benutzertransaktion greift also niemals mit einem Zugriffszeitpunkt t_Z^{be} , der kleiner als ein zuvor übermittelter Wert ist, auf das Backend zu. Aus dem Minimum aller übermittelten Zugriffszeitpunkte lässt sich so ermitteln, welche Versionen noch zugreifbar sind. Nicht mehr zugreifbare Versionen können gelöscht werden. Werden Benutzertransaktionen nur im Backend ausgeführt, ohne dass ein Cache involviert ist, muss der kleinste BOT aller laufenden Transaktionen in die Analyse aufgenommen werden.

Auf dem Cache gelten die Standard-Regeln für die Löschung von Versionen. Das heißt, dass dort nur das Minimum über alle BOTs von laufenden Transaktionen herangezogen wird, um zu entscheiden, welche Versionen zu löschen sind.

5.7 Middleware-basierter Zugriff auf ältere Snapshots

In der derzeitigen Implementierung unseres Prototyps (ACCACHE) wird der Zugriff auf ältere Snapshots durch einen speziellen Transaktionspool simuliert. Dieser basiert auf der Idee, dass nach dem Abschluss einer Transaktion, die den Datenbankzustand verändert hat, ein Pool von nebenläufigen Transaktionen⁵ angelegt wird, deren Startzeitpunkte direkt aufeinander folgen (z. B. $BOT_{BE}(T_1^{be}) = 10$, $BOT_{BE}(T_2^{be}) = 11$ usw.). Dabei wird sichergestellt, dass, während ein solcher Pool angelegt wird, keine andere Transaktion ihr Commit ausführt. Die Transaktionen innerhalb eines Pools greifen also alle auf den gleichen logischen Snapshot zu. Jeder Pool erhält eine Nummer, die gleichzeitig als Zugriffszeitpunkt dient und im WS an den Cache übermittelt wird. Wenn ein Cache eine lokale Backend-Transaktion anlegen will, referenziert er den entsprechenden Pool über den mitgelieferten Zustandszeitpunkt. Aus diesem bekommt er dann eine Transaktion zugewiesen. Ist der Pool von Transaktionen erschöpft, ist der Snapshot für den Cache nicht mehr erreichbar. Der Cache wird gezwungen, weitere WSs zu übernehmen, um einen neueren Pool referenzieren zu können. Wenn die Initialisierung einer lokalen Backend-Transaktion T_1^{be} fehlschlägt (z. B. wenn der Pool leer ist), wird die Benutzertransaktion T_1 einfach abgebrochen.

6 Fazit

Der vorliegende Aufsatz zeigt, dass der Einsatz eines (verteilten) Mehrversionenverfahrens für das CbDBC zwingend erforderlich ist, um einer Benutzertransaktion stets einen transaktionskonsistenten Zustand anbieten zu können. Damit lesende Zugriffe bevorzugt werden, wurde die Isolationsstufe Snapshot Isolation angestrebt. Dabei ist die im Aufsatz entwickelte Art der Nebenläufigkeitskontrolle bestens auf die Eigenschaften des CbDBC abgestimmt. Lesende Zugriffe werden niemals blockiert, ein Erstellen von Read Sets ist nicht erforderlich, die Replikate in den Caches lassen sich verzögert aktualisieren und es gibt keinerlei zusätzliche Kommunikation zwischen Cache und Backend. Besonders diese letztgenannte Eigenschaft ist wichtig. Der Cache ist sogar in speziellen Fällen in der Lage, eine Benutzertransaktion alleine, ohne Involvierung des Backends durchzuführen. Der erfolgreiche Abschluss einer Transaktion kann allein vom Backend verifiziert werden, sodass ein 2PC-Protokoll nicht notwendig ist. Durch Sondermaßnahmen lässt sich starke Snapshot Isolation erreichen, wobei der Cache oft selbst verifizieren kann, ob die zu ändernden Sätze vollständig im Cache vorliegen.

Das Poolen von Transaktionen ermöglicht, falls erforderlich, den Middleware-basierten Zugriff auf ältere Snapshots, wobei eine Integration dieser Funktionalität in ein bestehendes Datenbanksystem (z. B. PostgreSQL) anzustreben ist. Einerseits sind dadurch die älteren Zustände durch beliebig viele Benutzertransaktionen zugreifbar und andererseits ist ein deutlich besseres Leistungsverhalten zu erwarten.

⁵Durch diesen Trick können wir später bei Bedarf noch mehrfach einen existierenden Snapshot nutzen.

7 Kritik und Ausblick

Der einzige echte Nachteil der vorgestellten Nebenläufigkeitskontrolle liegt darin, dass der Backend-Zugriff von Benutzertransaktion auf den Zustand zurückgesetzt wird, der im Cache als letztes übernommen wurde. Der Cache bestimmt somit durch die Geschwindigkeit, mit der er WSs übernimmt, wie stark sein Zustand vom aktuellen Zustand im Backend abweicht. Dieser Abstand sollte nur gering sein, da sonst die Gefahr von Schreibkonflikten enorm steigt, was zum Abbruch vieler Transaktionen führen würde. Zur Verhinderung einer solchen Situation kann der Cache die Ausführung von Benutzertransaktionen künstlich verlangsamen, um mit höherer Priorität WSs einzuspielen. In der Regel ist jedoch zu erwarten, dass bei hoher Leselast im Gesamtsystem (60-80%) Konsistenz und Vollständigkeit in den Caches nur wenig vom aktuellen Datenbankzustand abweichen.

Die Eigenschaften der vorgestellten Synchronisation (Konfliktrate, Leistung, Skalierbarkeit) müssen durch Messungen noch genau untersucht werden. Dies stellt den Hauptanteil weiterer Forschungsbemühungen dar. Wünschenswert wäre dabei eine Integration in ein bestehendes Datenbanksystem (z. B. PostgreSQL). Außerdem müssen die Ideen und Methoden, um starke Snapshot Isolation zu garantieren, noch umgesetzt werden.

Literatur

- [ABK⁺03] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh und Berthold Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *VLDB*, Seiten 718–729, 2003.
- [APTP03] Khalil Amiri, Sanghyun Park, Renu Tewari und Sriram Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. In *ICDE*, Seiten 821–831, 2003.
- [BBG⁺95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil und Patrick O’Neil. A critique of ANSI SQL isolation levels. In *SIGMOD ’95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, Seiten 1–10, 1995.
- [BDD⁺98] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan, Jr., James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski und Mohamed Ziauddin. Materialized Views in Oracle. In *VLDB*, Seiten 659–664, 1998.
- [BG81] Philip A. Bernstein und Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.
- [BHM06] Andreas Bühmann, Theo Härder und Christian Merker. A Middleware-Based Approach to Database Caching. In Y. Manolopoulos, J. Pokorný und T. Sellis, Hrsg., *ADBIS 2006*, LNCS 4152, Seiten 182–199, Springer, 2006.
- [CRF08] Michael J. Cahill, Uwe Röhm und Alan D. Fekete. Serializable Isolation for Snapshot Databases. In *SIGMOD ’08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Seiten 729–738, 2008.
- [DS06] Khuzaima Daudjee und Kenneth Salem. Lazy Database Replication with Snapshot Isolation. In *VLDB 2006: Proceedings of the 32nd International Conference on Very Large Data Bases*, Seiten 715–726, 2006.

- [Fek09] Alan Fekete. Snapshot Isolation. In *Ency. of Database Systems*, Seiten 2659–2664, 2009.
- [FLO⁺05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O’Neil, Patrick O’Neil und Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.
- [GHOS96] Jim Gray, Pat Helland, Patrick E. O’Neil und Dennis Shasha. The Dangers of Replication and a Solution. In *SIGMOD Conference*, Seiten 173–182, 1996.
- [GLRG04a] Hongfei Guo, Per-Åke Larson, Raghu Ramakrishnan und Jonathan Goldstein. Relaxed Currency and Consistency: How to Say “Good Enough” in SQL. In Gerhard Weikum, Arnd Christian König und Stefan DeBloch, Hrsg., *SIGMOD*, Seiten 815–826. ACM, 2004.
- [GLRG04b] Hongfei Guo, Per-Ake Larson, Raghu Ramakrishnan und Jonathan Goldstein. Support for Relaxed Currency and Consistency Constraints in MTCache. In *SIGMOD*, Seiten 937–938, New York, NY, USA, 2004. ACM.
- [HB07] Theo Härder und Andreas Bühmann. Value Complete, Column Complete, Predicate Complete – Magic Words Driving the Design of Cache Groups. *VLDB Journal*, Seiten 805–826, 2007.
- [KB09] Joachim Klein und Susanne Braun. Optimizing Maintenance of Constraint-Based Database Caches. In *ADBIS*, Seiten 219–234, 2009.
- [Kle10] Joachim Klein. Concurrency and Replica Control for Constraint-based Database Caching. In *ADBIS*, Seiten 305–319. Springer, 2010.
- [LAK10] LAKSHYA Solutions Ltd. CSQL - Der Full-Table-Cache für MySQL, PostgreSQL und Oracle - Dokumentation, 2010.
<http://www.csqldb.com/prod-Documentation.html>.
- [LGZ04] Per-Åke Larson, Jonathan Goldstein und Jingren Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *ICDE*, Seiten 177–189. IEEE Computer Society, 2004.
- [LKPMJP05] Yi Lin, Bettina Kemme, Marta Patiño-Martínez und Ricardo Jiménez-Peris. Middleware-based Data Replication providing Snapshot Isolation. In *SIGMOD*, Seiten 419–430, 2005.
- [Mic10] Microsoft. Microsoft SQL Server 2008 R2 Dokumentation, 2010.
<http://www.microsoft.com/sqlserver/>.
- [Ora10] Oracle. Oracle 11g R2 Dokumentation, 2010. <http://www.oracle.com/technetwork/database/enterprise-edition/documentation/index.html>.
- [Pos10] PostgreSQL Global Development Group. PostgreSQL Dokumentation, 2010.
<http://www.postgresql.org/docs/>.
- [The02] The TimesTen Team. Mid-tier Caching: The TimesTen Approach. In *SIGMOD*, Seiten 588–593, 2002.
- [WK05] Shuqing Wu und Bettina Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In *ICDE*, Seiten 422–433, 2005.

A generalized join algorithm

Goetz Graefe

Hewlett-Packard Laboratories

Abstract

Database query processing traditionally relies on three alternative join algorithms: index nested loops join exploits an index on its inner input, merge join exploits sorted inputs, and hash join exploits differences in the sizes of the join inputs. Cost-based query optimization chooses the most appropriate algorithm for each query and for each operation. Unfortunately, mistaken algorithm choices during compile-time query optimization are common yet expensive to investigate and to resolve.

Our goal is to end mistaken choices among join algorithms by replacing the three traditional join algorithms with a single one. Like merge join, this new join algorithm exploits sorted inputs. Like hash join, it exploits different input sizes for unsorted inputs. In fact, for unsorted inputs, the cost functions for recursive hash join and for hybrid hash join have guided our search for the new join algorithm. In consequence, the new join algorithm can replace both merge join and hash join in a database management system.

The in-memory components of the new join algorithm employ indexes. If the database contains indexes for one (or both) of the inputs, the new join can exploit persistent indexes instead of temporary in-memory indexes. Using database indexes to match input records, the new join algorithm can also replace index nested loops join.

Results from an implementation of the core algorithm are reported.

1 Introduction

Non-procedural queries and physical data independence both enable and require automatic query optimization in a SQL compiler. Based on cardinality estimation, cost calculation, query rewrite, algebraic equivalences, plan enumeration, and some heuristics, query optimization chooses access paths, join order, join algorithms, and more. In most cases, these compile-time choices are appropriate, but poor choices often cause poor performance, dissatisfied users, and disrupted workflows in the data center. Investigation and resolution of intermittent performance problems are very expensive.

Our research into robust query processing has led us to focus on poor algorithm choices during compile-time query optimization. In order to avoid increasing complexity and sophistication during query optimization, e.g., by run-time feedback and statistical learning [MLR 03], our efforts center on query execution techniques.

The new join algorithm introduced here is a result of this research. Its design goal is a viable single replacement for all three traditional join algorithms by matching the performance of the best traditional algorithm in all situations. If both join inputs are sorted, the new algorithm must perform as well as merge join. If only one input is sorted, it must perform as well as the better of merge join and hash join. If both inputs are unsorted, it must perform as well as hybrid hash join. If both inputs are very large, it must perform as well as hash join with recursive partitioning or merge join and external merge sort with multiple merge levels. Finally, if one input is small, the new join algorithm must perform as well as index nested loops join exploiting a temporary or permanent index for the large input.

Table 1 summarizes the input characteristics exploited by index nested loops join, merge join, hybrid hash join, and the new join algorithm. Rather than merely performing a run-time choice among the traditional join algorithms, it combines elements from these algorithms and from external merge sort. Therefore, we call it “generalized join algorithm” or abbreviated “g-join.”

	INLJ	MJ	HHJ	GJ
Sorted input(s)		+		+
Indexed input(s)	+			+
Size difference			+	+

Table 1. Join algorithms and exploited input properties.

With one or two sorted inputs, g-join avoids run generation and merging, instead exploiting the sort orders in the inputs. For indexed inputs, it exploits the index either as a source of sorted data or as a means of efficient search.

For unsorted inputs, g-join employs run generation quite like external merge sorts for a traditional merge join. Unlike external merge sort, g-join avoids all or most merge steps, even leaving more runs than can be merged in a single merge step. Like hybrid hash join, it divides memory into two regions, one for immediate join processing and one for handling large inputs. If the size of the small join input is similar to the memory size, most memory is assigned to the first region; if the size of the small input is much larger, most or all memory is assigned to the second region. As in hybrid hash join, the size of the large join input does not affect the division of memory into regions.

The following sections review the traditional join algorithms (Section 2) and then introduce g-join (Section 3). Algorithm details for unsorted inputs of various input sizes and unknown input sizes (Section 4) are followed by answers for the “usual questions” about any new join algorithm or new algorithmic variant (Section 5). Based on those details and answers, replacement of the traditional join algorithms is discussed in depth (Section 6). Two partial prototype implementations permit an initial performance evaluation of g-join (Section 7). The last section offers our summary and conclusions from this effort so far.

2 Prior work

G-join competes with the well-known (index) nested loops join, merge join, and (hybrid) hash join algorithms, which are reviewed in detail elsewhere [G 93]. The diag-join algorithm [HWM 98] can serve as preprocessing step for most join algorithms including g-join. Moreover, the merge algorithm of g-join might seem similar to the diag-join algorithm as both exploit sorting and a buffer pool with sliding contents. The algorithms differ substantially, however, because diag-join only applies in the case of foreign key integrity constraint whereas g-join is a general join algorithm, because diag-join depends on equal insertion and scan sequences whereas g-join does not, and because diag-join is inherently heuristic whereas g-join guarantees a complete join result.

In addition to join algorithms, prior research has investigated access paths, in particular index usage – from covering indexes (also known as index-only retrieval) to index intersection (combining multiple indexes for the same table) and query execution plans with dynamic index sets [MHW 90]. Inasmuch as such access plans require set operations such as intersection, g-join serves the purpose; otherwise, source data access in tables and indexes is not affected by g-join.

Prior research also has investigated join orders in the contexts of dynamic programming [SAC 79], very large queries [IK 90], and dynamic join reordering [AH 00, BBD 05, LSM 07]. Most of those research directions and their results are orthogonal to g-join and its relationship with the traditional join algorithms.

The following sections assume a join operation with an equality predicate between the two join inputs. Special cases such as joining a table with itself, joining on hash values, etc. are feasible but ignored. Similarly, we ignore join operations without equality predicates.

3 The new join algorithm

G-join is a new algorithm; it is not a run-time switch among algorithms, e.g., the traditional join algorithms. It is based on sorted data and thus can exploit sorted data stored in b-tree indexes as well as sorted intermediate results from earlier operations in the query execution plan. For unsorted inputs, it employs run generation very similar to a traditional external merge sort. Thereafter, it avoids most or all of the merge steps in a traditional merge sort. Moreover, the behavior and cost function of recursive and hybrid hash join have guided the algorithm design for unsorted inputs. Nonetheless, g-join is based on merge sort and is not a variant of hash join, e.g., partitioning using an order-preserving hash function. Delaying all details to subsequent sections, the basic idea is as follows.

For unsorted inputs, run generation produces runs like an external merge sort, but merging these runs can be omitted in most cases. Any difference in the sizes of the two join inputs is reflected in the count of runs for each input, not in the sizes of runs.

With sufficient memory and a sufficiently small number of runs from the smaller input, join processing follows roughly (but not precisely) the sort order of join key values. A buffer pool holds pages of runs from the small input. Pages with higher keys successively replace pages with lower keys. A single buffer frame holds pages of runs from the large input (one page at a time) while such pages are joined with the pages in the buffer pool. In other words, during join processing, most memory is dedicated to the small input and only a single page is dedicated to the large input. With respect to memory management, the buffer pool is reminiscent of the in-memory hash table in a hash join, but its contents turns over incrementally in the order of join key values.

If merging is required, the merge depth is kept equal for both inputs. This is rather similar to the recursion depth of recursive hash join and quite different from a merge join with two external merge sorts for inputs of different sizes. For fractional merge depths – similar to hybrid hash join – memory is divided into segments for immediate join processing and for buffers for temporary files. Thus, g-join requires memory and I/O for temporary files in very similar amounts as recursive and hybrid hash join. Even bit vector filtering and role reversal are possible, as are integration of aggregation and join operations.

If one or both inputs are sorted, run generation and merging can be omitted. With two sorted inputs, the algorithm “naturally simplifies” to the logic of a traditional merge join. If the small input is sorted, the buffer pool holds only very few pages, very similar to the “back-up” logic in merge join with duplicate key values. If the large input arrives sorted, g-join joins its pages with the buffer pool contents by strictly increasing join key values.

If one or both of the inputs is indexed, g-join exploits available indexes in its merge logic. If one input is tiny and the other input is huge, the merge logic skips over most data pages in the huge input, thus mimicking traditional index nested loops join. If the small input is indexed and the index can be cached in memory, there is no need for sorting the large input – like hash join as well as the non-traditional mode of index nested loops join.

4 Unsorted inputs

Many of the algorithm's details can best be explained case by case. This section focuses on inner joins of two unsorted, non-indexed inputs with practically no duplicate key values, with uniform (non-skewed) key value distributions, and with pages holding multiple records and thus non-trivial key ranges. Later sections relax these assumptions.

As the size of the small input is crucial to achieving join performance similar to recursive and hybrid hash join for unsorted inputs, the discussion divides cases by the size of the small input relative to the memory size. In all cases, the large input may be larger than the small input by a few percent or by orders of magnitude. The core algorithm that most other cases depend on is covered in Section 4.2.

In order to explain g-join step-by-step, Sections 4.1 through 4.5 assume accurate a priori knowledge of the size of input R . Section 4.6 relaxes this assumption.

The following descriptions assume that compile-time query optimization anticipated that input R will be smaller than input S . Therefore, input R is consumed first and drives memory allocation and run generation.

The memory allocation is M (pages or units of I/O) and the maximal fan-in in merge steps as well as the maximal fan-out in partitioning steps is F . F and M are equal except for their units and a small difference due to a few buffers required for asynchronous I/O etc. If M is around 100 or higher, this difference is not significant and usually ignored.

During merge steps, read operations in run files require random I/O. Large units of I/O (multiple pages) are a well-known optimization for external merge sort and for partitioning, e.g., during hash join and hash aggregation. The same optimization also applies to the runs and to the merge operations described in 4.2.1.

4.1 Case $R \leq M$

The simplest case is a small input that fits in memory, i.e., $R \leq M$. No run generation is required in this case. Instead, g-join retains R in memory in an indexed organization (e.g., a hash table, but it may be a b-tree), and then processes the pages of S one at a time.

With all temporary files avoided, the I/O cost of g-join in this case is equal to that of traditional in-memory hash join. When using the same in-memory data structure, the CPU effort of the two join algorithms is also the same.

4.2 Case $R = F \times M$

The next case is the one in which Grace hash join [FKT 86] and hybrid hash join operate in the same way, with F pairs of overflow files, no immediate join processing during the initial partitioning step, and all memory required during all overflow resolution steps.

In this case, g-join creates initial runs from both inputs R and S . With replacement selection for run generation, the number of runs from input R is $F/2 + 1$. Even if the number of runs from input S is much larger than the maximal merge fan-in F , no merging is required. Instead, inputs R and S are joined immediately from these runs. Practically all memory is dedicated to a buffer pool for pages of runs from input R . Input S requires only a single buffer frame as only one page at a time is joined with the contents of the buffer pool.

4.2.1 Page operations

Careful scheduling governs read operations in runs from inputs R and S . At all times, the buffer pool holds some key range of each run from input R . The intersection of those

key ranges is the “immediate join key range.” If the key range in a page from input S falls within the immediate join key range, the page is eligible for immediate join processing.

The schedule focuses on using, at all times, the minimal buffer pool allocation for pages of the small input R. It grows its memory allocation only when necessary in order to process pages from the large input S one at a time, shrinks the buffer pool as quickly as possible, and sequences pages from the large input S for minimal memory requirements.

The minimal memory allocation for the buffer pool requires one page for each run from input R. Its maximal memory allocation depends on key value distributions in the inputs, i.e., distribution skew and duplication skew. With uniform key value distributions and moderate amounts of duplicate key values, about two pages for each run from input R should suffice. Two pages for each of $F/2+1$ runs amount to M pages. In other words, g-join can perform the join immediately after run generation in this case, independently of the size of input S. Thus, if indeed two pages per run from input R suffice, memory needs and I/O effort of g-join match those of hash join.

Algorithm overview

Figure 1 illustrates the core algorithm of g-join. Various pages (double-ended arrows) from various runs cover some sub-range of the domain of join key values (dotted horizontal arrow). Some pages of runs from input R are resident in the buffer pool (solid arrows) whereas some pages have already been expelled or not yet been loaded (dashed arrows). The pages in the buffer pool define the immediate join key range (dotted vertical lines). It is the intersection of key ranges of all runs from input R. Some pages of runs from input S are covered by the immediate join key range (solid arrows) whereas some have already been joined or cannot be joined yet (dashed arrows). Differently from the diagram, there usually are more runs from input S than from input R. Again, at any one time, memory holds multiple pages of each run from input R but only one page from one of the runs from input S.

In Figure 1, the buffer pool contains 2 pages each from runs 1 and 3 of input R and 3 pages from runs 2. In the illustrated situation, the next action joins a page from run 2 of input S with the pages in the buffer pool, whereupon the buffer pool may shrink by a page from run 2 of input R.

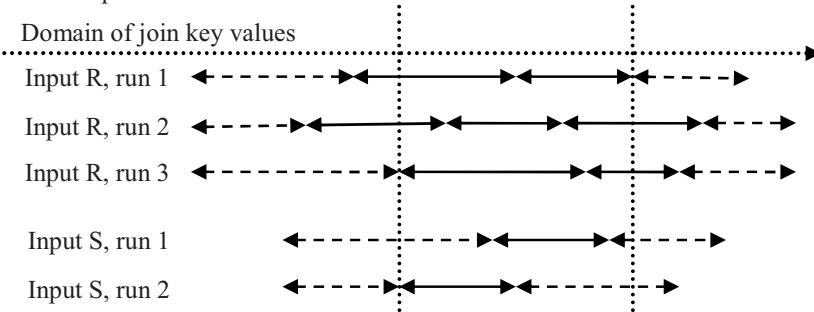


Figure 1. Runs, pages, buffer pool, and the immediate join key range.

Data structures

The immediate join key range expands and contracts as it moves through the domain. Multiple priority queues guide the schedule of page movements. These priority queues require modifications when the buffer pool reads or drops pages of input R and when a page

of S joins with the pages in the buffer pool. All priority queues are sorted in ascending order such that top entry holds the smallest key value. The priority queues are named A through D:

- A. This priority queue guides how the buffer pool grows. Each run from the small input R has one entry in this priority queue at all times. Its top entry indicates the next page to load into memory. The sort key is the high key of the newest page in the buffer pool for a run.
- B. This priority queue guides how the buffer pool shrinks. Each run from input R has one entry in this priority queue at all times. Its top entry indicates the next page to drop from memory. The sort key is the high key of the oldest page in the buffer pool for a run.
- C. This priority queue guides how the buffer pool shrinks. Each run from input S has an entry in this priority queue. Its top entry indicates the next page to join from the large input S . The sort key is the low key value in the next page on disk.
- D. This priority queue guides how the buffer pool grows. Each run from input S has an entry in this priority queue. Its top entry indicates the next page to schedule from input S . The sort key is the run with the high key value in the next page on disk.

Priority queue A is similar in function and size ($F/2$ entries) to a priority queue guiding page prefetch (“forecasting”) in a traditional external merge sort. Priority queue B could have an entry for each page in memory rather than for each run from input R . In that case, it would be similar in function and size (M entries) to a priority queue used for page replacement in a buffer pool. Priority queues C and D are similar in size ($S/(2M)+1$ entries) to that of a priority queue guiding a traditional merge sort to merge in each step the smallest ones among all remaining runs, which is the fastest way to reduce the number of runs.

Priority queues C and D employ information from pages not yet read during the join process. With a very moderate loss in predictive precision, priority queue C can use the highest key value already seen instead of the lowest future key value.

Finally, it is possible to omit priority queue D and schedule pages of input S entirely based on priority queue C. This algorithm variant does not require a larger maximal buffer pool, although it may require a large buffer pool over longer periods.

Algorithm

The algorithm initializes the buffer pool and priority queues A and B with the first page of each run from input R . Priority queues C and D initially hold information about the first page of each run from input S . The algorithm continues step by step until all pages of all runs from input S have been joined, i.e., priority queues C and D are both empty.

Each step tests the top entries of priority queues C and then D whether they can be joined immediately. If so, it reads the appropriate page of input S and joins it to the pages of input R in the buffer pool. It then replaces the page of input S in priority queues C and D with the next page from the same run from input S . If the end of the run is reached, the run is removed from priority queues C and D. If the replaced page used to be the top entry of priority queue C, the buffer pool may drop some pages, guided by priority queue B.

Otherwise (if the top entries of priority queues C and D cannot be joined immediately), the buffer pool grows by loading some additional pages from input R . Priority queue A guides this growth until the top entry in priority queue D can be joined immediately.

The overall complexity of the priority queue operations is modest: each page in all runs from inputs R and S goes through precisely two priority queues. Replace and erase

operations are required in these priority queues. Tree-of-loser priority queues [K 73] can implement these operations with a single leaf-to-root pass.

The actual I/O operations as well as operations on individual records will likely exceed the CPU cost for the priority queues. Operations on individual records include insertion and deletion in an in-memory index when pages of input R enter and exit the buffer pool as well as search operations in this index to match records of input S. These insertions, deletions, and searches are similar in cost and complexity to the equivalent operations in a hash join and its in-memory hash table.

Prototype implementation

Our first prototype implementation of the g-join logic employs priority queue B to guide shrinking the buffer pool. It does not merge key values into an in-memory b-tree index. The prototype has run-time switches that control whether or not priority queue D is used and whether priority queue C is sorted on the highest value already seen or the lowest value not yet seen. Priority queue D is not used in any of the experiments reported below.

For a uniform distribution of join key values and a uniform distribution of run sizes, it requires about two pages of input R, as discussed in detail later. Two inputs with 100 and 900 runs of 1,000 pages, i.e., a total of 1,000,000 pages, can be processed in the priority queues in less than 1 second using a laptop with a 2.8 GHz CPU. Clearly, 1,000,000 I/O operations take more time by 3-4 orders of magnitude. Maintenance of the priority queues takes a negligible amount of time.

4.2.2 Record operations

As pages of runs from input R enter and exit the buffer pool, their records must be indexed in order to permit fast probing with join key values from records of input S.

A local index per page of input R is not efficient, as each record of input S would need to probe as many indexes as there are pages in the buffer pool. This can substantially increase the cost of probing compared to a global index for all records in the buffer pool.

A global index must support not only insertions but also deletions. After a new page has been read into the buffer pool, its records are inserted into the global index; before a page is dropped from the buffer pool, its records are removed from the global index.

The implementation of the global index can use a hash table or an ordered index such as a b-tree. B-tree maintenance can be very efficient if the b-tree contains only records in the key range eligible for immediate join processing. For efficient insertion, the runs from input R can be merged (as in a traditional merge sort) and then appended to the b-tree index. For efficient deletion, entire leaf pages can be removed from the b-tree.

On the other hand, a hash table may support more efficient join processing than a b-tree index. Even if every record in input R eventually joins with some records of input S, each page of input S join with only a few records of input R. Thus, a lot of skipping and searching is required in a global b-tree index.

Hash table implementations with efficient insertion and deletion therefore seem the most appropriate data structure for in-memory join processing, i.e., the buffer pool with records from input R with individual pages of runs from input S.

4.3 Case $M < R < F \times M$

This case falls between the prior two cases, i.e., the case in which hybrid hash join shines. During the initial partitioning step of hybrid hash join, some memory serves as out-

put buffers for the partition overflow files and some memory holds a hash table and thus enables immediate join processing. If the small input is only slightly larger than the available memory allocation, most of the join result is computed during the initial partitioning step and only a small fraction of both join inputs incurs I/O costs. If the small input is much larger than the available memory allocation, hardly any join result is computed immediately and a large fraction of both join inputs spills to overflow partitions.

G-join also employs a hybrid of two algorithms. It divides memory among them and employs the same division of memory as hybrid hash join. A fraction of memory equal to the size of the hash table in hybrid hash join enables immediate join processing as in Section 4.1. Thus, while consuming the join inputs for the first time, g-join computes the same fraction of the join result as hybrid hash join. The remaining memory enables run generation quite similarly to the algorithm of Section 4.2.

The less memory run generation uses, the smaller the resulting runs are. The goal is to produce $F/2$ runs from input R, because the algorithm of Section 4.2 can process $F/2$ runs in a single step. Interestingly, the required formulas for the memory division are equal in hybrid hash join and g-join.

Specifically, hybrid hash join requires at least K overflow partitions and output buffers, with K derived as follows. K partitions can hold $K \times M$ pages from input R. This memory allocation leaves $M - K$ pages for the hash table and immediate join processing. In order to process all input in a single step, input R must fit within the hash table plus these partitions, i.e., K must satisfy $R \leq (M - K) + K \times M$. Solving for K gives $K \geq (R - M) \div (M - 1)$ or $K = \text{ceiling}((R - M) \div (M - 1))$.

G-join uses the same division of memory as hybrid hash join. Immediate join processing uses $M - K$ pages and K pages are used for preparation of temporary files. In g-join, these pages serve as workspace for run generation. Thus, $R - (M - K)$ pages are the input to run generation, with $R - (M - K) \leq K \times M$ because $R \leq (M - K) + K \times M$.

With replacement selection using a workspace of K pages, the average run size is $2K$. The resulting count of runs is $(K \times M) / (2K) = M/2$. This is precisely the number of runs from the smaller input R that can be processed by the algorithm described in Section 4.2.

As in hybrid hash join, immediate in-memory join processing must be assigned a specific set of key values. In hybrid hash join, an appropriate range of hash values is assigned to the in-memory partition. G-join can employ a similar hash function or it can simply retain the lowest key values from input R. For the latter variant, the required data structure and algorithm is very similar to that of an in-memory “top” algorithm [CK 97], i.e., a priority queue. This design choice is best with respect to producing sorted output suitable for the next operation in the query execution plan.

While g-join consumes input R, it employs a priority queue to determine the key range eligible for immediate join processing. While g-join consumes input S, it employs a hash table for join processing. The hash table contains precisely those records that remained in the priority queue after consuming input R.

In summary, g-join running in hybrid mode divides memory like hybrid hash join, retains the same fraction of the smaller input R in memory, performs the operations required for in-memory just as efficiently as hybrid hash join, and produces nearly sorted output.

4.4 Case $R = F^2 \times M$

In this case, hash join requires precisely two partitioning levels. Assuming a perfectly uniform distribution of hash values, two partitioning levels with fan-out F produce over-

flow files from input R equal in size to the available memory, enabling in-memory hash join for each partition. Suitable overflow partitions from input S require the same two partitioning levels, independent of the size of input S and its partition files.

G-join similarly moves each input record through two temporary files. After run generation produces $R \div (2M) = (F^2 \times M) \div (2M) = F^2/2$ initial runs for input R, a single merge level with merge fan-in F reduces the count of runs to $F/2$. Input S also goes through two levels, namely run generation and one level of merging. Thereafter, the algorithm of Section 4.2 applies, independent of the size of input S and its number of remaining run files.

4.5 Case $F \times M < R < F^2 \times M$

In this case, a hash join requires more than one partitioning level but less than two full partitioning levels. The partial level is realized by hybrid hash join when joining partitions produced by the initial partitioning step.

G-join first aims to produce $F^2/2$ runs from input R by dividing memory similar to the algorithm in Section 4.3. If the size of R is close $F \times M$, most memory is used for immediate join processing during this phase. If the size of R is close to $F^2 \times M$, very little memory is used for immediate join processing and most memory is used as workspace for run generation. More specifically, the calculation $K = \text{ceiling}((R - M) \div (M - 1))$ of Section 4.3 is replaced with $K = \text{ceiling}((R/F - M) \div (M - 1))$ to account for one additional merge level.

After this initial hybrid step, g-join merges all runs once, reducing the number of runs from input R by a factor of F to $F/2$. All runs from input S are also merged once with a fan-in F . The final step applies the algorithm of Section 4.2 to the remaining runs.

4.6 The general case

The preceding descriptions assume precise a priori knowledge of the size of input R. Dropping this assumption, the following discussion assumes that actual run-time sizes are not known until the inputs have been consumed by the join algorithm, that input R is consumed before input S, and that R is smaller than S. Should S be smaller than R, role reversal is possible after run generation for both inputs but it is not discussed further.

In order to calibrate expectations, it is worthwhile to consider the behavior of hybrid hash join under these assumptions. The preceding discussions of hybrid hash join assume perfectly uniform distributions of hash values. For a perfect assignment of hash values to the in-memory hash table and to the overflow partitions, hybrid hash join also requires prior knowledge of the desired size of the in-memory hash table, i.e., of the precise size of the build input. Without this knowledge, hash join loses some degree of efficiency. Different designs and implementations of hash join suffer in different places. In all cases, changing the size of the in-memory hash table and its hash buckets is quite complex.

G-join, with two unsorted inputs of unknown sizes, first consumes the input anticipated to be the smaller one. If that input R fits in memory (case $R \leq M$), run generation for input S can be avoided entirely, and g-join performs similarly to an in-memory hash join.

Otherwise, the algorithm divides memory between immediate join processing and run generation. With an unknown input size, the best current estimate is used. This estimate may change over time, and the memory allocation is adjusted accordingly. Note that such an adjustment is easily possible in g-join.

The most conservative variant of g-join prepares for two huge inputs, i.e., run generation uses all available memory. If the first input turns out to be small and fit in memory, run

generation for the second input is skipped in favor of immediate join processing. Otherwise, run generation for both inputs is completed. In this variant, g-join performs rather like Grace hash join [FKT 86] without dynamic de-staging [NKT 88].

The memory allocation at the end of consuming input R is preserved throughout run generation and immediate join processing for input S. After run generation for input S, if one of the two inputs has produced no more than $F/2$ runs, final join processing can commence immediately without any intermediate merge steps.

Otherwise, runs from the smaller input are merged until $F/2$ runs remain. Each merge step merges the smallest remaining runs, which reduces the number of remaining runs with the least effort [K 73]. Due to the effects of replacement selection, this will most likely affect the first and last initial runs, because the sizes of all other runs tend to be similar to the sizes of these two runs together. If run generation produces precisely $F/2+1$ runs, merging the first and last runs produces $F/2$ runs of similar size.

The merge policy also attempts to minimize the size of the largest run of input R left for final join processing. Thus, it might be useful to perform multiple merge steps with moderate fan-in rather than one merge step with maximal fan-in, even if doing so requires merging slightly more than the minimal data volume.

Next, g-join merges runs from the larger input. Again, each merge step merges the smallest remaining runs. Even with no other merge steps, it might be useful to merge the first run and the last run produced during run generation. In fact, it is often possible to merge the first and last runs immediately after the end of the input, i.e., while the last run is being formed. Merging continues until the smallest remaining run is at least as large as the largest remaining run from the smaller input. For unsorted inputs, this stopping condition leads to equal merge depth for both inputs. For join inputs of very different sizes, this is a crucial performance advantage of g-join when compared to merge join, very similar to the main advantage of hash join over merge join.

The crucial aspect is not the count of runs but their sizes. Ideally, the runs from input S should be of similar size as the runs from input R. More specifically, the smallest run of input S should be at least as large as the largest run of input R. Assuming reasonably uniform distributions of join key values, this ensures that a buffer pool of M pages suffices to join $F/2$ runs from input R with any number of runs from input S, which is the final step in g-join for unsorted inputs of unknown size.

4.7 Summary for unsorted inputs

In summary, g-join processes two unsorted inputs about as efficiently as recursive hybrid hash join. This is true for input sizes from tiny to huge and for both known and unknown input sizes. In particular, g-join exploits inputs of very different sizes by limiting the merge depth for both inputs to that required by the smaller input, quite similar to the recursion depth in hash join. Moreover, g-join is able to divide memory between immediate join processing and preparation of temporary files, very similar to hybrid hash join in both memory allocation and performance effects.

G-join is based on sorting its inputs rather than on hash partitioning. It even produces the join result roughly in sorted order such that it might be useful in subsequent operations within the query execution plan. This and similar questions are discussed in the following section, and the issue whether g-join can substitute for the traditional join algorithms is considered thereafter.

5 The usual questions

This section discusses skew in the distribution of join key values, binary operations of the relational algebra other than inner join, complex queries with multiple join operations, and parallel query execution.

5.1 Skew

Skew can affect the performance of g-join in several ways. For example, extreme duplication of a single key value in the small input may temporarily force a very large buffer pool. A temporary file might be required, comparable to a buffer pool in virtual memory rather than real memory. In those extreme cases, both hash join and merge join effectively resort to nested loops join, usually using some form of temporary file and repeated scans.

In general, a buffer pool extended by virtual memory or an equivalent technique is one of two “water proof” methods for dealing with extreme cases of skew. The other one reduces both inputs R and S to a single run and then performs a merge join. Short of these methods, however, a variety of techniques may reduce the impact of skew on the performance of g-join. The following describes some of those.

Run generation may gather some statistics about the range and distribution of key values in each run. If skew is an issue, the merge step may process inputs R and S just a bit further than discussed so far. As a result, input R will have fewer than $F/2$ runs remaining and the available memory allocation can support more than two buffer pool pages per run. Input S will have larger runs with a smaller key range per page, thus requiring fewer pages of input R in the buffer pool during join processing.

Even in the case of uniform distributions of join key values, merging runs from input S until the smallest run from input S is twice as large or even larger than the largest run from input R reduces the buffer pool requirements. Again, the key ranges in each page of input R and in each page of input S are crucial. If the pages from input S have a smaller key range on average, fewer runs from input R require multiple pages in the buffer pool at a time.

Rather than merging entire runs, it is also possible read individual pages from input S twice. If the buffer pool is at its maximal permissible size, cannot be shrunk, and no pages can be joined immediately, joining the low key range of some pages from input S might enable shrinking the buffer pool and then growing it again to extend the immediate join key range. Priority queue C can track key ranges already joined. If the key value in priority queue C falls in the middle of a page rather than a page boundary, the page must be read again to complete its join with the buffer pool and input R.

5.2 Beyond inner joins

In addition to inner joins, traditional join algorithms also serve semi-joins and anti-semi-joins (related to “in” and “not in” predicates with nested queries) as well as outer joins (preserving rows without matches from one or both inputs). In fact, some of the joins permit some optimizations. For example, a left semi-join can avoid the inner loop in nested loops join, avoid back-up logic in merge join, and short-circuit the loop traversing a hash bucket in hash join. On the other hand, some join algorithm require additional data structures. For example, a right semi-join implemented as nested loops join needs to keep track of which rows in the inner input have already had matches from earlier outer rows, and a hash join needs an additional bit with each record in its hash table.

In addition to join operations, relational query processing employs set operations such as intersection, union, and set difference. These operations may be specified in the query syntax or they may be inserted into a query execution plan during query optimization, in particular in plans exploiting multiple indexes for a single table. For example, conjunction queries might employ two indexes and intersect the resulting sets of row references.

G-join supports all of these operations. For some of them, it requires an additional bit for each record from the first input while a record is resident in the buffer pool. All other decisions for left and right semi-join, anti-semi-join, and outer join can readily be supported with small changes and optimizations in the join processing logic.

5.3 Complex queries

A join method is useful for database query processing only if it passes the “Guy Lohman test” [G 93]. It must be useful not only for joining two inputs but also in complex query execution plans that join multiple inputs on the same or on different columns.

In complex query execution plans with multiple join operations, g-join can operate as pipelined operation (in particular with pre-sorted inputs) or as “stop-and-go” operation or “pipeline breaker” for one or both inputs. The choice is dictated by input sizes or by external control, e.g., from the query optimizer. For example, a pipeline break can avoid resource contention with an earlier or a later operation in the query execution plan, it can enable a later operation to improve its resource management based on more accurate estimates of the join output size, or it can enable general dynamic query execution plans.

The output of g-join is almost sorted. If a perfect sort order is desired, the sort can be optimized to take advantage of the guaranteed key range. At any point in time, g-join can produce output only within a certain range of join key values defined by the current contents of the buffer pool. While the join output within that key range fits into the memory allocation of the sort operation, the sort operation can avoid temporary run files and immediately pipeline its output to the next operation in the query execution plan. Even if temporary run files are required, they can be merged eagerly up to a key value defined by the key range in the buffer pool of g-join.

If two instances of the new join form a producer-consumer relationship within a query execution plan and thus pipeline the intermediate result from one to the other, and if the join columns in the two join predicates share a prefix (or ideally are entirely the same), the output order produced by the first join improves the performance of the second. Even if the intermediate result is not perfectly sorted, its ordering has a high correlation with the required ordering in the second join operation. Thus, run generation in the second join operation achieves longer intermediate runs, fewer runs, and thus less intermediate merge effort or a smaller buffer pool during final join processing.

For this effect, it is not required that the join columns in the two join predicates be equal. It is sufficient that they share a prefix. If so, longer runs and thus more efficient join processing is entirely automatic. While this is also true for merge join with explicit sort operations, exploiting equal join predicates requires hash teams [GBC 98], which are more complex than traditional binary hash join algorithms but relatively simple compared to generalized hash teams [KKW 99] that exploit partial overlap of join predicates.

In relational data warehouses with star schemas around one or more fact tables, star joins combine multiple small dimension tables with a very large fact table. Optimizations for star joins include star indexes (b-tree indexes for the fact table with row identifiers of dimension tables as search keys), semi-join reduction (semi-joins between dimension tables

and secondary indexes of the fact table), and Cartesian products (of tiny dimension tables). It appears that g-join can support all required join operations and in fact exploits the size difference in joins of small (dimension) tables and large (fact) tables as well as hash join.

5.4 Parallel query execution

Parallel query execution relies on partitioning a single intermediate result, on pipelining intermediate results between operations in producer-consumer relationships, or on both. G-join can participate in all forms of parallel query execution. Partitioning intermediate results into subsets is entirely orthogonal to the choice of local algorithms. Pipelining intermediate results might be aided by exploiting not only equal column sets in join predicates of neighboring operations but also by exploiting join predicates that merely share a prefix. In other words, there is reason to expect that g-join enables efficient pipelining more readily than multiple merge join operations with intermediate sort operations. Compared to query execution plans with multiple hash join operations, g-join enables similar degrees of pipelining but it does so making much better use of the sort order of intermediate results.

6 Replacing traditional join algorithms

It is unrealistic to expect that g-join will displace all traditional join algorithms rapidly. Even if this goal succeeds eventually, it will take many years. As an analogy, it has taken decades for hash join to be implemented in all products. On the other hand, slow adoption permits additional innovation beyond the initial ideas. For example, after hybrid hash join was first published in 1984, Microsoft SQL Server included hash join only in 1998 [GBC 98], but it also included hash teams to mirror the advantages of interesting orderings in query execution plans based on merge join. Nonetheless, even if it is unrealistic to propose or to expect a rapid adoption of g-join, it seems worthwhile to make the case for replacing the traditional join algorithms.

6.1 Hash join

Hash join offers advantages over the other traditional join algorithms for unsorted, non-indexed join inputs. Thus, the focus of this discussion must be the case of unsorted, non-indexed input, e.g., intermediate results produced by earlier operations in the query.

Throughout Section 4, the cost of the new algorithm mirrors the cost of hash join including recursive partitioning and hybrid hash join. In addition to a fairly similar cost function for unsorted inputs, g-join also produces nearly sorted output without any extra effort.

The traditional optimizations of hash join readily transfer to g-join. For example, if compile-time query optimization errs in estimating relative input sizes, role reversal after run generation for both inputs is trivial. Similarly, due to separate phases consuming the two join inputs, bit vector filtering readily applies to g-join.

Hash join can readily integrate aggregation (grouping) on the join column, albeit only for its build input. If aggregation is desired for the inputs of g-join, sorting must merge the affected input until it forms a single run. Thus, some efficiency is lost, more so when aggregation applies to the larger input than for the smaller input, which g-join merges to $F/2$ runs in any case. Note that Section 4.2 proposes merging these $F/2$ runs to form an in-memory b-tree index. This merge step with a single output stream can readily perform aggregation or duplicate elimination for input R.

Hash teams are another optimization for query execution plans based on hash join and hash aggregation. They mirror the effects of interesting orderings in plans based on merge join and stream-aggregation. Earlier sections already discuss the ability of g-join to produce, consume, and exploit interesting orderings of intermediate results.

In summary, hash join and its optimizations shine for unsorted, non-indexed inputs. G-join closely matches the performance of hash join and its optimizations in all cases. While the performance of g-join does not exceed that of hash join, it produces and consumes sorted intermediate results and it eliminates the danger of a mistaken choice among multiple join algorithms.

6.2 Merge join

Merge join shines when both join inputs are sorted by prior operations, e.g., join or aggregation operations on the same columns or by scans of b-tree indexes. In those cases, g-join exploits the sorted inputs. Run generation is omitted and join processing consumes the join inputs, which take on the roles of runs in the discussion of Section 4. The buffer pool requires one or two pages for the smaller input. Note that a traditional merge join requires a small buffer pool to back up its inner scan in the case of duplicate join key values. In other words, if both inputs are sorted, g-join operates very much like a traditional merge join and its underlying movement of pages in the buffer pool.

If only one join input is sorted by prior operations, g-join consumes it as a single run and performs run generation for the other input, similar to run generation as discussed in Section 4 for two unsorted inputs. The performance of g-join in this case matches or exceeds that of merge join, because merging the unsorted input may stop early when many runs remain. The performance also matches or exceeds that of hash join, because no effort is required for partitioning or merging the input already sorted.

In summary, g-join matches or exceeds the performance of merge join in all cases. Note that qualitative information such as the sort order of indexes, scans, and intermediate results is known reliably at compile-time or at least at plan start-up-time; the decision whether or not sorting is required does not depend on error-prone quantitative information such as cardinality or cost estimation.

6.3 Index nested loops join

Index nested loops join shines in two distinct cases. First, when the outer input including an index fits in memory, the resulting algorithm is rather like an in-memory hash join. Second, if there is an index for the large inner input and there are fewer rows (or distinct join key values) in the outer input than pages in the inner input, then index nested loops join avoids reading useless pages in the large inner input. In both of these cases, g-join can match the performance of index nested loops join.

In the first case, run generation stops short of writing records from input R to temporary run files. Instead, all records remain in the run generation workspace, which takes on the role of the buffer pool. In-memory join processing may use an in-memory index structure like in-memory hash join or order-based merge logic like merge join.

In the second case, which is the traditional case for index nested loops join, g-join sorts the small input and then performs a zigzag merge join of the two inputs, i.e., the merge logic attempts to skip over useless input records rather than scan over them, and it applies this logic in both directions between the join inputs. If the number of distinct join key val-

ues in the smaller input is lower than the number of pages in the larger input, many of these pages are never needed for join processing. This is, of course, precisely the effect and the performance advantage of index nested loops join over merge join and hash join, and g-join mirrors this performance advantage precisely.

In order to achieve full performance, index access should to be optimized with proven techniques such as asynchronous prefetch, pinning page on the most recent root-to-leaf path in the buffer pool, leaf-to-root search using cached boundary keys, etc. These techniques limit the I/O cost of an index nested loops join to that of scanning the two indexes involved.

It is important to note that all required choices – whether to sort or to rely on the sort order of the input, whether to build an in-memory index or rely on a database index – are based on schema information, not on cardinality estimation. In other words, the detrimental effects of errors in compile-time cardinality estimation are vastly reduced.

7 Performance

A prototype of the core algorithm produced the results reported here. Michael Carey and his students are building a query execution system at UC-Irvine that includes g-join.

G-join combines well-studied elements of prior query processing algorithms. Implementation techniques and behavior of run generation, replacement selection, merging, in-memory hash tables, index creation, index search, etc. are all well understood. A new implementation of those algorithmic components is unlikely to yield new insights or results.

The principal novel component and the core of g-join is the schedule of page movements during join processing, i.e., the technique described in Section 4.2. The buffer pool loads and drops pages of runs from input R while individual pages of runs from input S are scheduled, read, and joined with the buffer pool contents. This is the algorithm component modeled in detail in the prototype. Actual I/O operations with on-disk files and operations on individual records are not included in the prototype.

The crucial performance characteristic that is not immediately known from prior work is the required size of the buffer pool. In the best case, only a single page of each run from input R is required; in the worst case, the buffer pool must grow to hold all of input R. The expectation from the discussion above is that about 2 pages per run from input R are required in the steady state. If this expectation is accurate, the I/O volume of g-join is practically equal to that of recursive and hybrid hash join. This assumes, of course, an unsorted input for g-join (as g-join would exploit pre-sorted inputs) and a perfectly uniform distribution of hash values in hash join (which is required to achieve balanced partitioning and to match the standard cost function in practice).

7.1 Implementation status and baseline experiment

The prototype focuses on page movements in the algorithm of Section 4.2. Input parameters include the run count for each input (defaults 10 and 90 runs), the page counts for each input (default 40 pages per run), and the number of values in the domain of join key values (default 1,000,000 distinct values). With random key ranges in input pages, the run sizes are only approximate. The output includes the average and maximum buffer pool sizes, and may include a trace showing how the buffer pool grows and shrinks over time.

With the default values, the prototype simulates a join of input R with about 400 pages to input S with about 3,600 pages. Figure 2 illustrates the size of the required buffer pool

for input R over the course of an experiment. The x-axis indicates how many pages of the large input S have already been joined. The y-axis shows the size of the buffer pool at that time, indicated as the average number of pages per run from input R. It can be clearly seen that this size hovers around 2 pages per run from input R. The buffer pool repeatedly grows beyond that, but not by very much. The maximum in this experiment is 2.3 pages per run (equal to 23 pages total in this experiment). The buffer pool also shrinks below 2 pages per run repeatedly and in fact more often and more pronounced than growing beyond 2 pages per run. At the end of the join algorithm, the buffer pool size shrinks to 1 page per run.

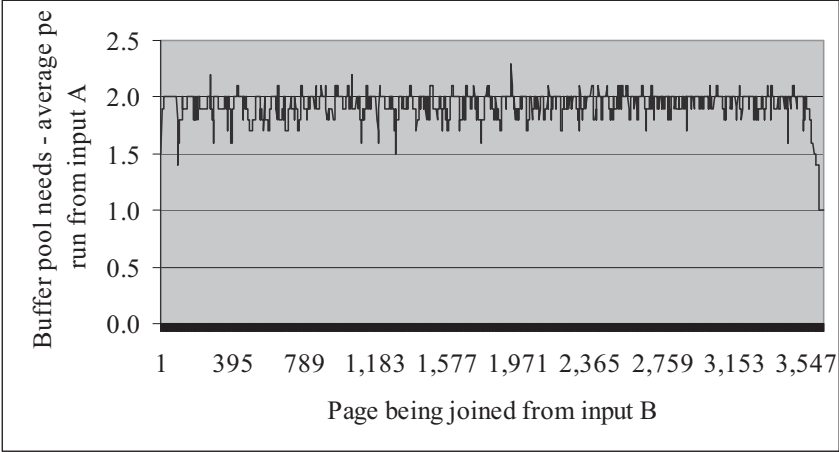


Figure 2. Buffer pool requirements over time.

7.2 Run counts and sizes

The next experiment shows how 2 pages per run from input R is quite stable across a range of memory sizes and input sizes.

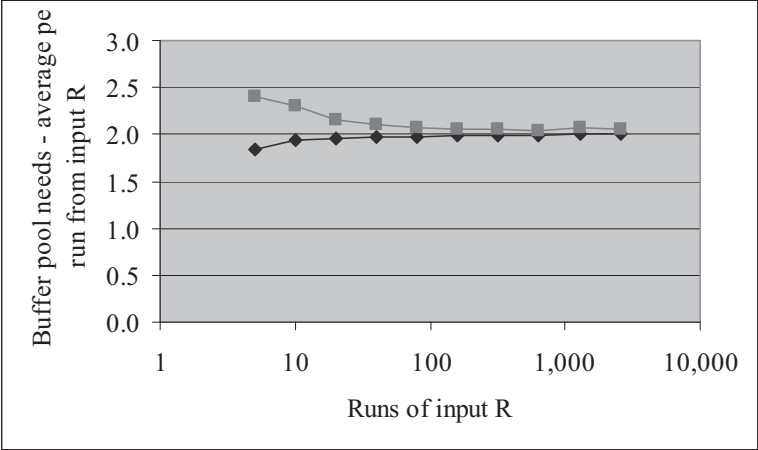


Figure 3. Buffer pool requirements with varying memory and input sizes.

Specifically, memory sizes in this experiment range from 10 pages to 5,120 pages, varied by powers of 2. Run sizes are assumed twice the memory size. The number of runs from input R is half the memory size (such that the buffer pool holds 2 pages per run). The number of runs from input S is 9 times larger, as in the prior experiment. Thus, run sizes vary from 20 to 10,240 pages and run counts vary from 5 to 2,560 for input R and from 45 to 23,040 for input S. Thus, input sizes vary from 100 to 26 million pages for input R and from 900 pages to 236 million pages for input S.

Figure 3 relates the number of runs from input R and the buffer pool requirements, both the average (lower curve) and the maximal (upper curve) buffer pool size for each memory and input size. In all cases, each run from the smaller input R requires about 2 pages in the buffer pool, confirming the basic hypothesis that g-join perform similar to hash join for large, unsorted inputs.

With an increasing number of runs from each input, the average grows closer to 2 and the maximum shrinks closer to 2. The former is due to many runs from the large input S; some page in some run from input S spans any page boundary in the runs from input R, and thus all runs from input R require about 2 pages in the buffer pool at all times. The latter is due to many runs from the small input R; even while some run might need 3 instead of 2 pages for a short period, it has little effect on the number of buffer pool pages when divided by the number of runs from input R. Thus, while the number of buffer pool pages is usually below 2, it sometimes is above 2, but only by a little bit and only for a short time.

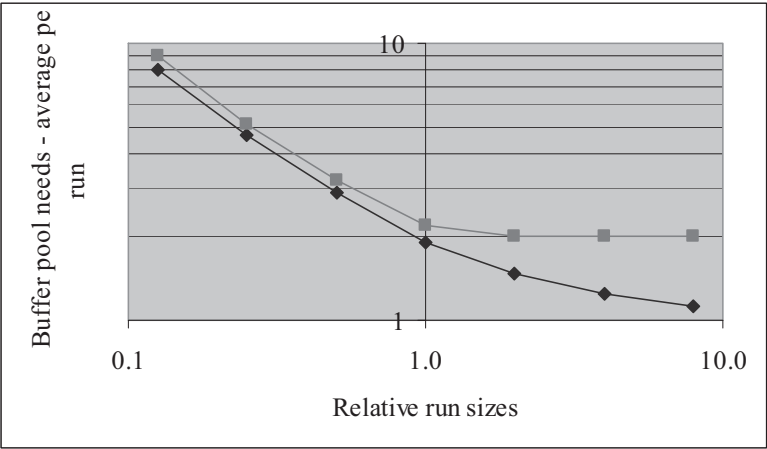


Figure 4. Buffer pool requirements with different run sizes.

Figure 4 illustrates the effect of insufficient or excessive merging in the large input S. In all cases, all runs from input R are of the same size and all runs from input S are of the same size. The x-axis indicates the quotient of runs size from input S to those from input R. The left-most data points indicate runs from input R 8 times larger than those from input S; the right-most data points indicate runs from input S 8 times larger than those from input R. The y-axis, ranging from 1 to 10 on a logarithmic scale, again shows average and maximal buffer pool needs, with the total buffer pool size divided by the number of runs. For all data points, there are 10 runs from input R and 90 runs from input S.

In the left half of the diagram, it is readily apparent that g-join needs many buffer pool pages per run if runs from input S are smaller. This is due to the large key range covered by

each page in such a run: it takes many pages of a larger run from input R to cover such a key range. In the right half of the diagram, where runs from input S are larger than the runs from input R, the average buffer pool requirements shrink almost to 1 page per run from input R. The maximal buffer pool requirements, however, do not.

Figure 4 permits two conclusions. First, in order to minimize buffer pool requirements, runs from input S require merging until all remaining runs are larger than all runs from input R. In this mode of operation, the cost function of g-join for unsorted inputs most closely resembles that of hash join. Second, if buffer space is readily available for runs from input R, it can be exploited to save some effort merging runs from input S. For example, with 10 buffer pool pages for each run from input R, runs from input S may be left smaller than those from input R, thus saving merge effort for input S.

7.3 Skew

Just like hash join suffers from skew in the distribution of hash values, g-join may suffer from various forms of skew in its inputs. There are several forms of skew, e.g., the sizes of runs (due to dynamic memory allocation during run generation) as well as skew in key value distribution. The form of skew most likely to affect the performance of g-join is skew in the sizes of runs. Such skew might be due to dynamic memory management during run generation or a correlation between input order and desired sort order.

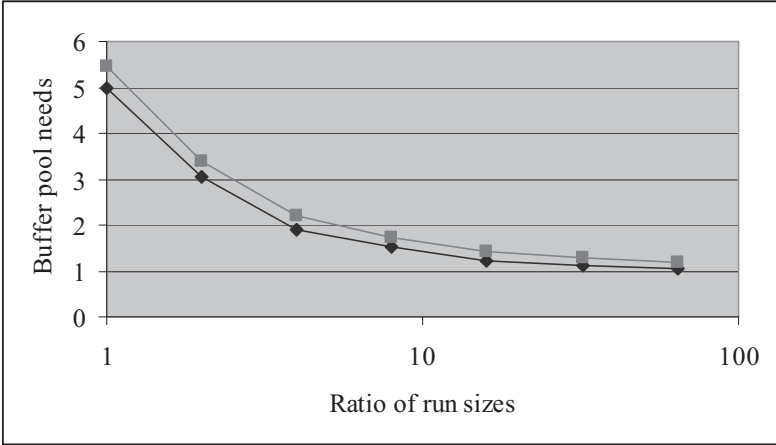


Figure 5. Buffer pool requirements with varying run sizes.

Figure 5 illustrates the effect when runs from the same input differ in size. (In Figure 4, all runs from either one input are the same size.) In Figure 5, sizes for runs from input R are chosen from the range 400 to 3,200 pages, i.e., the largest and smallest run might differ by a factor 8. Sizes for runs from input S might also differ by a factor 8, but the range is chosen differently for each data point. For the left-most data point (ratio = 1), the range is also 400 to 3,200 pages; for the right-most data point, the range is 64 times larger or 25,600 to 204,800 pages.

The buffer pool needs are governed by the largest run from input R and the smallest run from input S. They are equal for the central data point (ratio = 8), and the average and maximal buffer pool requirement for each run from input R is about 2 pages. It is actually less because some runs from input R are small and some runs from input S are large.

At the left-most data point, some runs from input R are much larger than some runs from input S. Those runs require many more pages in the buffer pool, and in fact dominate the overall buffer pool requirements. The number of pages per run from input R (about 5) is almost equal to the ratio of runs sizes (about 8).

At the right-most data point, all runs from input R are much smaller than all runs from input S. Thus, each page from input R covers many pages from input S. With fairly small key ranges within pages from input S, only a few runs from input R require 2 pages in the buffer pool at any point in time. Thus, the maximum buffer pool size (divided by the number runs from input R) is approaching the ideal value of 1.

7.4 Hyrax experiences

Michael Carey and students at UC-Irvine have experimented with g-join [L 10] within their Hyrax research prototype. Their implementation differs from the original design described above by using quicksort for run generation rather than replacement selection. Thus, runs are equal in size to the memory allocation, not twice. More importantly, this algorithm choice exacerbates the problem of a last run much smaller than memory and with a key range per page much larger than in other runs. They also observe that incidental ordering in an input has little effect on run sizes and run counts, which of course is different with replacement selection.

Their experiments so far show faster random writes during hash partitioning than random reads during merging and join processing in g-join. This is most likely due to automatic write-behind in hash join (using additional system memory) and the lack of forecasting and asynchronous read-ahead in this implementation of g-join. Nonetheless, their experiments confirm the above observations about the number of I/O operations and the amount of data written to and read from temporary files.

Finally, their experiments show average and maximal buffer pool sizes larger than shown in the experiments above, but still consistently below 3 pages if runs of input S are no smaller than runs of input R. It has been impossible to reproduce these larger buffer pool sizes with the initial implementation of the core algorithm used in the experiments reported above.

8 Summary and conclusions

In summary, the new, generalized join algorithm (“g-join”) combines elements of the three traditional join algorithms yet it is an entirely new algorithm. This is most obvious in the case of two unsorted inputs, where g-join performs run generation like an external merge sort but then joins these runs without merging them (or with very little merging even for very large inputs). Therefore, g-join performs like merge join in the case of two sorted inputs and like hash join in the case of two unsorted inputs, including taking advantage of different input sizes. Our partial prototype implementation and our experimental evaluation confirm the analytical performance expectations.

In the case of indexed inputs, g-join exploits the indexes for sorted scans or even for searches in a zigzag merge join. Skipping over many pages in the index and fetching only those input pages truly required for the join is the main advantage of index nested loops join over hash join and merge join. G-join mirrors this advantage by using a zigzag merge join (skipping forward) rather than a traditional merge join (scanning forward). Thus, g-join performs as well as index nested loops join for a large, indexed, inner join input.

In conclusion, we believe that g-join competes with each of the three traditional join algorithms where they perform best. It could therefore be a replacement for each or for all of them. Replacing all three traditional join algorithms with g-join eliminates the danger of mistaken (join) algorithm choices during compile-time query optimization. Thus, g-join improves the robustness of query processing performance without reducing query execution performance.

Acknowledgements

Mike Carey, Guangqiang “Aries” Li, and Vinayak Borkar have implemented a variant of g-join and compared its performance with their implementation of hash join. Barb Peters, Harumi Kuno, and the reviewers suggested numerous improvements in the presentation of the material. Stephan Ewen (TU Berlin), Stefan Krompass, and Wey Guy provided excellent feedback on the algorithm, including test cases for robustness and general stress tests.

References

- [AH 00] Ron Avnur, Joseph M. Hellerstein: Eddies: continuously adaptive query processing. SIGMOD 2000: 261-272.
- [BBD 05] Pedro Bizarro, Shivnath Babu, David J. DeWitt, Jennifer Widom: Content-based routing: different plans for different data. VLDB 2005: 757-768.
- [CK 97] Michael J. Carey, Donald Kossmann: On saying "enough already!" in SQL. SIGMOD 1997: 219-230.
- [FKT 86] Shinya Fushimi, Masaru Kitsuregawa, Hidehiko Tanaka: An overview of the system software of a parallel relational database machine GRACE. VLDB 1986: 209-219.
- [G 93] Goetz Graefe: Query evaluation techniques for large databases. ACM Comput. Surv. 25(2): 73-170 (1993).
- [GBC 98] Goetz Graefe, Ross Bunker, Shaun Cooper: Hash joins and hash teams in Microsoft SQL Server. VLDB 1998: 86-97.
- [HWM 98] Sven Helmer, Till Westmann, Guido Moerkotte: Diag-join: an opportunistic join algorithm for 1:N relationships. VLDB 1998: 98-109.
- [IK 90] Yannis E. Ioannidis, Younkyung Cha Kang: Randomized algorithms for optimizing large join queries. SIGMOD 1990: 312-321.
- [K 73] Donald E. Knuth: The art of computer programming, Volume III: sorting and searching. Addison-Wesley 1973.
- [KKW 99] Alfons Kemper, Donald Kossmann, Christian Wiesner: Generalised hash teams for join and group-by. VLDB 1999: 30-41.
- [L 10] Guangqiang Li: On the design and evaluation of a new order-based join algorithm. MS-CS thesis, UC Irvine, (2010).
- [LSM 07] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, Guy M. Lohman: Adaptively reordering joins during query execution. ICDE 2007: 26-35.
- [MHW 90] C. Mohan, Donald J. Haderle, Yun Wang, Josephine M. Cheng: Single table access using multiple indexes: optimization, execution, and concurrency control techniques. EDBT 1990: 29.
- [MLR 03] Volker Markl, Guy M. Lohman, Vijayshankar Raman: LEO: An autonomic query optimizer for DB2. IBM Systems Journal 42(1): 98-106 (2003).
- [NKT 88] Masaya Nakayama, Masaru Kitsuregawa, Mikio Takagi: Hash-partitioned join method using dynamic destaging strategy. VLDB 1988: 468-478.
- [SAC 79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access path selection in a relational database management system. SIGMOD 1979: 23-34.

View Maintenance using Partial Deltas

Thomas Jörg and Stefan Dessloch
University of Kaiserslautern, Germany
{joerg|dessloch}@cs.uni-kl.de

Abstract: This paper addresses maintenance of materialized views in a warehousing environment, where views reside on a remote database. We analyze so called Change Data Capture techniques used to capture changes (also referred to as deltas) at the source systems. We show that many existing CDC techniques do not provide complete deltas but rather incomplete (or partial) deltas. Traditional view maintenance techniques, however, require complete deltas as input. We propose a generalized technique that allows for maintaining a class of materialized views using partial deltas.

1 Introduction

Materialized views are used to pre-compute (intermediary) query results to speed up query evaluation [GM95]. Upon updates to the base data, materialized views need to be maintained to regain consistency. Assuming that updates affect just a small part of the base data, it seems wasteful to maintain a view by recomputing it from scratch. It is often more efficient to compute only the changes required to update the view. This approach is referred to as incremental view maintenance.

The concept of materialized views has been applied in distributed environments, where base tables and materialized views reside on different machines connected by a network [ZGMHW95, ZGMW98, AASY97, AAM⁺02]. A machine hosting materialized views is usually called data warehouse (DWH). In a DWH environment, views are typically maintained in a deferred manner, i.e. deltas are gathered at the sources and propagated periodically in batches.

It has been shown that traditional view maintenance techniques can be applied in DWH environments. However, since global transactions are prohibitively expensive, special care must be taken w.r.t. synchronization. Previous research focused on this aspect only (cf. Sec. 6).

Our work addresses an orthogonal problem. In a DWH environment, so called Change Data Capture (CDC) techniques are used to gather deltas at the source systems [KC04]. The captured deltas are often partial (or incomplete). Partial deltas may lack attribute values – the initial state of an updated tuple may not be available, for instance. Furthermore, the type of partial deltas may be uncertain, i.e. inserted tuples may not be distinguishable from updated ones.

The reasons for deltas being partial are twofold. First, there are CDC approaches that cannot deliver non-partial (or complete) deltas due to principal restrictions. Second, the

CDC process may become more efficient if partial deltas are acceptable. Traditional view maintenance techniques, however, require complete deltas and cannot be used in such an environment. In this paper we give answers to the following questions: Is it possible to maintain materialized views using partial deltas? How can traditional view maintenance techniques be generalized such that partial deltas can be propagated?

The remainder of this paper is organized as follows. In Sec. 2 we give an overview of CDC techniques used in practice. We explain why many CDC techniques provide partial deltas. Traditional view maintenance techniques presume the availability of non-partial deltas and thus, cannot be applied in many DWH setups. Hence, we aim at generalizing these techniques such that partial deltas can be propagated. To this end, we propose a formal model for partial deltas in Sec. 3. This model will provide a basis for our generalized update propagation approach. In Sec. 4 we discuss techniques to apply partial deltas to materialized views. In Sec. 5 we first review a view maintenance algorithm by Griffin et al. based on algebraic differencing, which provides the basis for our work. We then discuss view maintainability in the context of partial deltas. As we will see, views cannot be maintained using partial deltas in general. However, we will identify an important class of views, which we will call dimension views, that is maintainable here. In the remainder of this section, we show how the Griffin et al. algorithm can be generalized for the propagation of partial deltas. We discuss related work in Sec. 6 and conclude in Sec. 7.

2 Change Data Capture

Change Data Capture (CDC) is a general term for techniques that gather change information (or deltas) at source systems [KC04]. We analyzed existing CDC modules and identified four main approaches, namely utilization of audit columns, log-based CDC, change tracking, and computing snapshot differentials.

Audit columns: Source systems may maintain dedicated columns (so called audit columns) to store timestamps or version numbers for individual tuples. Whenever a tuple is changed, it is assigned a fresh timestamp. Audit columns can serve as selection criteria to retrieve tuples that have been updated since the last CDC cycle.

In its most simplistic form, a single audit column is appended to each base table. A new timestamp is assigned whenever a tuple is either inserted or updated. Hence, insertions cannot be distinguished from updates when deltas are extracted. To work around this limitation, two audit columns can be appended to base tables. The first audit column is used to store the time of insertion while the second stores the time of the last update. However, deletions remain undetected when tuples are physically deleted. Tuples can instead be logically deleted by adding yet another audit column to store the time of the (logical) deletion. However, CDC techniques backed by audit columns are generally unable to capture the initial state of updated tuples. This is obvious considering that updates are performed in-place and previous values are overwritten.

Log-based CDC: Source systems may keep a log of changes that is appended in the event of an update. Several implementation approaches for log-based CDC exist: If the source system provides active database capabilities such as triggers, deltas can be written to dedicated log tables. Log-based CDC can also be implemented by means of application logic. Database log scraping is another common CDC approach. The idea is to exploit the transaction logs kept by the database system for backup and recovery. Deltas can be extracted using database-specific utilities.

Log-based CDC mechanisms are generally capable of providing complete deltas. However, their efficiency can be improved if partial deltas are acceptable. For view maintenance the so called net effect of changes is required as input. To obtain the net effect, the change log needs to be post-processed. When a tuple has been changed multiple times, the effects of these changes are combined to produce a single delta tuple. If a tuple has been inserted and subsequently updated, for instance, a delta tuple of type insertion with the updated values is produced.

The net-effect computation is more efficient if partial output deltas are acceptable. The following quote has been taken from the SQL Server 2008 documentation on the change capture feature [Mic].

Because the logic to determine the precise operation for a given change adds to query complexity, this option is designed to improve query performance when it is sufficient to indicate that [...] the change is either an insert or an update, but it is not necessary to explicitly distinguish between the two.

Note that not being able to distinguish between insertions and updates means that the initial state of updated tuples is also not available.

Change Tracking: Change Tracking is an alternative change capture feature of SQL Server 2008 built into the database engine [Mic]. Change tracking is being advertised as light-weight change capture solution that offers better scalability than audit column or trigger-based solutions.

Change tracking is done by making a note of the primary key of the tuple that changed, along with the type of the change (insert, update, or delete) and a version number in an internal table. To retrieve deltas, the change tracking table needs to be joined to the corresponding base table, because it does not store any non-key attributes. More precisely, an outer join needs to be used, because deleted tuples are no longer found in the base tables. Thus, deltas produced by change tracking do not contain any information about deleted tuples except for the primary keys. Furthermore, the initial state of updated tuples cannot be reconstructed, because it has been overwritten in the base table.

Snapshot Differentials: Legacy and custom applications often lack a general purpose query interface. However, it is often possible to dump a system snapshot into the file system. Deltas can then be inferred by comparing successive snapshot files.

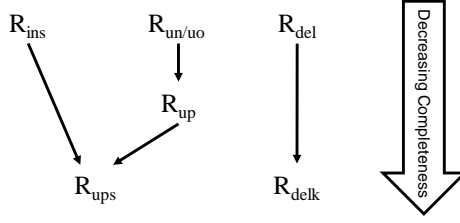


Figure 1: Delta sets with decreasing Completeness

In summary, several CDC approaches are used in practice and many of them produce partial deltas. Since there are CDC techniques that do not have this restriction, one could argue that only these techniques should be used for view maintenance in DWH environments. However, partial deltas can be captured more efficiently. Furthermore, source systems typically remain autonomous. Often, the system owners are reluctant to changes, thereby limiting the choice of practicable CDC techniques.

3 A Model for Partial Deltas

This section introduces a formal model for partial deltas. The analysis of CDC techniques in the previous section revealed different kinds of partial deltas that need to be captured in this model. First, it may not be possible to distinguish insertions from updates. Second, deltas may lack the initial state of updated tuples. Third, only the primary key of deleted tuples may be known.

Definition 1 (Partial Deltas) Let $R(pk, a)$ be a relation with primary key pk and a set of attributes a . Let R_{old} be the state of R before it is changed and R_{new} the state of R hereafter. Partial deltas are a six-tuple of sets $(R_{ins}, R_{un/uo}, R_{del}, R_{up}, R_{ups}, R_{delk})$ where

- $R_{ins} \subseteq R_{new}$ denotes a set of tuples inserted into R (referred to as insertions),
- $R_{del} \subseteq R_{old}$ denotes a set of tuples deleted from R (referred to as deletions),
- $R_{un/uo}(pk, a_{un}, a_{uo})$ with $\pi_{pk, a_{un}}(R_{un/uo}) \subseteq R_{new}$ and $\pi_{pk, a_{uo}}(R_{un/uo}) \subseteq R_{old}$ denotes a set of tuples updated in R (referred to as update pairs). The initial state and the current state of updated tuples is given by (pk, a_{uo}) and (pk, a_{un}) , respectively,
- $R_{up} \subseteq R_{new}$ denotes a set of tuples updated in R in their current state only (referred to as partial updates),
- $R_{ups} \subseteq R_{new}$ denotes a set of tuples either inserted or updated in R (referred to as upserts),

- $R_{delk} \subseteq \pi_{pk}(R_{old})$ denotes a set of primary keys of tuples deleted from R (referred to as partial deletions)

such that each change at the tuple level from R_{old} to R_{new} is reflected by exactly one tuple in one of the delta sets R_{ins} , R_{del} , $R_{un/uo}$, R_{up} , R_{ups} , or R_{delk} . That is, the primary key values are pairwise disjoint across the delta sets and

$$\pi_{pk}(R_{new} - R_{old}) = \pi_{pk}(R_{ins} \cup R_{un/uo} \cup R_{up} \cup R_{ups}) \text{ and}$$

$$\pi_{pk}(R_{old} - R_{new}) = \pi_{pk}(R_{del} \cup R_{un/uo} \cup R_{delk}).$$

Figure 1 depicts the connection between the six delta sets. For each change in R there is a delta tuple in one of the delta sets. However, a delta tuple may appear in different delta sets. The alternative placements are indicated by the arrows in Fig. 1. The completeness decreases while moving from the upper to the lower delta sets. Decreasing completeness means that either attribute values become unavailable (update pairs to partial updates and deletions to partial deletions) or the type of the change becomes uncertain (partial updates to upserts and insertions to upserts).

The CDC techniques introduced in the previous section, can be characterized using our model for partial deltas. Figure 2 depicts the delta sets provided by different CDC techniques.

4 Change Data Application

The purpose of change propagation is maintaining the (remote) materialized view to re-synchronize it with the source data. In this paper we show that change propagation is closed under the model for partial deltas (for a certain class of view definitions). That is, given partial input deltas the process of change propagation results in partial output deltas again, that comply with the model introduced in Sec. 3. The completeness of the output deltas may however differ from the completeness of the input deltas. The output deltas may contain upserts, for instance, even when none of the input delta sets did. Once the deltas have been propagated to the DWH, they are applied to the view. Depending on their completeness, different techniques can be used for delta application. An overview is provided in the following.

	ins	del	un/uo	up	ups	delk
Audit columns	✓	✓		✓		
Log-based CDC	✓	✓	✓			
Log-based CDC (efficient net-effect computation)		✓			✓	
Change Tracking	✓			✓		✓
Snapshot differentials	✓	✓	✓			

Figure 2: Delta sets provided by different CDC techniques

Insertions To apply insertions, the SQL interface provides the INSERT statement. Furthermore many databases are equipped with bulk loading facilities to insert larger batches of records in an efficient manner.

(Partial) deletions Assuming that the view has a primary key column, just key values are required to apply deletions (whereas attribute values are not). Thus partial deletions are sufficient for view maintenance in such cases. Non-partial deletions are however relevant for change propagation. In general, the resulting deltas are less partial when non-partial deletions are provided as input. The interrelationship will be examined in Sec. 5.

Partial updates Much like partial deletions, partial updates are sufficient for maintaining views with key columns.

Update pairs In contrast to partial updates, update pairs include the old state of updated tuples. To update a tuple in place, the old state is not required. However, the DWH often keeps historical data. Data historization is typically done using the so called Slowly Changing Dimensions technique [KR02]. To do so, it is important to understand which attributes have been changed. Given an update pair this can be found out easily. Given a partial update, however, a warehouse lookup is required to find out about the initial values. The latter approach is obviously less efficient.

Upserts To apply upserts, one can either attempt an UPDATE first and issue an INSERT if no rows were affected or else run an INSERT first and issue an UPDATE if the inserted key violates the uniqueness constraint. This method has been criticized as being rather inefficient [KC04]. In the latest SQL standard the MERGE statement has been introduced to work around this issue. MERGE can be used to insert or update tuples depending on whether a user-defined condition matches. While MERGE is more efficient than the former approach, it performs worse than a sequence of INSERT and UPDATE statements. However, the latter approach is only possible when inserts and updates are given in two distinct delta sets (i.e. deltas are less partial).

It is interesting to understand the relation between change capture and change application in the context of partial deltas: While more partial deltas can be captured more efficiently, the application of more partial deltas is less efficient. Thus, there is a trade-off between change capture and change application. Note that these steps are performed at distinct systems. It is thus possible to shift workload from the source systems to the DWH (by capturing more partial deltas) or vice versa (by capturing less or non-partial deltas).

5 View Maintenance using Partial Deltas

A number of approaches to incremental view maintenance have been proposed in literature (cf. [GM95] for an overview). Our work is based on an approach known as algebraic

V	ΔV	∇V
$\sigma_p(S)$	$\sigma_p(\Delta S)$	$\sigma_p(\nabla S)$
$\pi_A(S)$	$\pi_A(\Delta S) - \pi_A(S_{old})$	$\pi_A(\nabla S) - \pi_A(S_{new})$
$S \bowtie T$	$(S_{new} \bowtie \Delta T) \cup (\Delta S \bowtie T_{new})$	$(S_{old} \bowtie \nabla T) \cup (\nabla S \bowtie T_{old})$

Table 1: Delta Rules by Griffin et al.

differencing that was introduced in [KP81] and subsequently used for view maintenance in [QW91]. Some corrections to the minimality results of [QW91] and further improvements have been presented in [GLT97]. The basic idea is to differentiate the view definition to derive expressions that compute the change to the view without doing redundant computations.

The remainder of the section is structured as follows. In Sec. 5.1 we recall a conventional view maintenance algorithm by Griffin et al. [GLT97]. We proceed with a discussion on view maintainability in the context of partial deltas in Sec. 5.2. We will identify a class of views, which we call dimension views, that are maintainable in this context. The Griffin et al. algorithm uses so called delta rules to derive incremental expressions for view maintenance from view definitions. We propose generalized delta rules in Sec. 5.3. These rules allow for deriving incremental expressions to maintain views using partial deltas.

5.1 Algebraic Differencing for View Maintenance

In this section, we recall the algorithm proposed by Griffin et al. [GLT97] that provided the base for our work. Objects of interest are relations and relational expression presented in relational algebra. Relational expressions are used to define derived relations (or views). Changes to base relations are modeled as two sets – the set of deleted tuples and the set of inserted tuples. For a relation R the set of deleted tuples is denoted by ∇R and the set of inserted tuples is denoted by ΔR . Updates are not modeled explicitly but represented by delete-insert-pairs, i.e. for each update in R there is a corresponding delta tuple in ∇R and in ΔR .

Given a relational expression that defines a view, incremental expressions are derived by recursively applying so called delta rules. The delta rules¹ defined in [GLT97] are depicted in Tab. 1. From a relational expression V two incremental expressions ∇V and ΔV are derived that compute the deletions and insertions to the view, respectively. To this end, subexpressions in V that match the “patterns” shown in the left column of Tab.1 are recursively replaced by incremental counterparts found in the middle column or the right column to obtain ΔV or ∇V , respectively. Intuitively, the delta rules in Tab. 1 can be understood as follows.

- **Selection:** An inserted tuple is propagated through a selection, if it satisfies the filter

¹Since our work is focused on Select-Project-Join (SPJ) views, delta rules for union, intersection, and set difference have been omitted.

predicate. A deletion is propagated through a selection, if the tuple used to satisfy the filter predicate.

- **Projection:** An inserted tuple is propagated through a projection, if no alternative derivation existed before the change. A deletion is propagated through the projection, if no alternative derivation remains after the change.
- **Join:** New tuples appear in the join of two relations, if a tuple inserted into one relation joins to tuples in the other one. Tuples disappear from the join, if a tuple deleted from one relation used to join to tuples in the other one before the change.

The view maintenance algorithm by Griffin et al. requires complete change information. Thus, it cannot be applied if deltas are partial as described in Sec. 3. We propose a generalized view maintenance algorithm that gracefully deals with partial deltas for a restricted (but important) class of view definitions.

5.2 Dimension Views

In general, materialized views cannot be maintained using partial deltas. Consider the following example. Say there is a base relation $R(pk, a)$ with pk being the primary key column and a simple derived view $V(a) := \pi_a(R)$. Say we use a CDC mechanism that does not provide the initial state of updated tuples (such as audit columns or change tracking). Obviously, V cannot be maintained in case of an update to R . While it is straightforward to add the updated tuple to V , it is unclear which tuple in V needs to be discarded (or overwritten) in return. Similar considerations hold for the other kinds of partial deltas, i.e. partial deletions and upserts.

Note, that V was maintainable if it included the primary key column pk . Including primary keys is thus a necessary condition for views to be maintainable using partial deltas. However, not all primary keys from the source relations need to be retained in the view definition. We will discuss the selection of keys in the following. At first, we define a class of views, which we call dimension views, that has interesting properties w.r.t. maintenance using partial deltas.

Definition 2 (Dimension View) *Let V be a relational expression defining a view that contains projections, selections, and joins only. V is called dimension view, if each join $R \bowtie_p S$ has a join predicate of the form $(R.a = S.pk)$ where a is a (set of) attributes of R and pk is the (composite) primary key of S .*

In the subsequent sections, we will show that dimension views are maintainable using partial deltas, if they include those primary key attributes that are not functionally dependent on any other key attributes. With other words, all key attributes used in join predicates do not need to be included in the view.

Dimension views are commonly found in DWHs. While they are usually called dimension tables here, they store derived data and can thus be seen as views. DWHs typically

Cust _{old}				Addr _{old}		
CID	CName	CDiscount	CAddr	AID	ACity	ACountry
1	Adam	0%	1	1	Austin	US
2	Bob	0%	2	2	Berlin	DE
3	Carl	0%	3	3	Chemnitz	DE

Cust _{new}				Addr _{new}		
CID	CName	CDiscount	CAddr	AID	ACity	ACountry
1	Adam	5%	1	1	Aachen	DE
2	Bob	0%	4	2	Berlin	DE
4	Dave	0%	4	4	Dresden	DE

Figure 3: Sample base tables in the old and new state

use a star schema to store multi-dimensional data that consists of fact and dimension tables [KC04, KR02]. Dimension tables are used to join together data on business entities that originates from multiple source systems. For improved query performance, dimension tables are typically denormalized. Dimension tables include a unique identifier for business entities referred to as business key. Typically, no other keys originating from the sources are stored here. Note that these keys would be functionally dependent on the business key in the denormalized dimension table. Our work is thus directly applicable to incremental maintenance of dimension tables.

Example 1 Figure 3 depicts two relations that are going to be used as a running example throughout the paper. The *Cust* relation stores the ID, name, and discount of customers and a reference to an address, which is stored in the *Addr* relation. The idea is to derive a dimension table *D* from these base tables. Dimension tables are typically de-normalized. Consider the sample view definition.

$$D := \pi_{CID, CName, CAddr}(Cust) \bowtie_{(Cust.Addr = Addr.AID)} \sigma_{ACountry = 'DE'}(Addr)$$

The view is restricted to German customers, furthermore the customer discount column is dropped. Note that the view is a dimension view w.r.t. Def. 2.

5.3 A Generalized View Maintenance Algorithm

We propose a generalization of the algorithm by Griffin et al. that allows for maintaining dimension views using partial deltas. We proceed as follows: First, we explain how partial deltas can be represented by means of delete-insert sets used by the original algorithm. Second, we propose generalized delta rules for projection, selection, and join. We show that these operators are closed under the model for partial deltas. Third, we conclude that dimension views can be maintained by our algorithm.

View maintenance algorithms (including the one by Griffin et al.) model deltas by two sets – the set of deleted tuples and the set of inserted tuples. We are going to refer to this model as delete-insert delta model or delete-insert model for short. This model does not directly match our model for partial deltas introduced in Sec. 3. The latter uses a six-tuple

$$\begin{aligned}
\Delta R(pk, a) &:= R_{ins} \cup \pi_{pk, a_{un}}(R_{un/uo}) \cup R_{up} \cup R_{ups} \\
\nabla R(pk, a, flag) &:= \pi_{pk, a, comp}(R_{del}) \cup \pi_{pk, a_{uo}, comp}(R_{un/uo}) \cup \\
&\quad \pi_{pk, NULL, up}(R_{up}) \cup \pi_{pk, NULL, ups}(R_{ups}) \cup \pi_{pk, NULL, up}(R_{delk}) \\
&\quad \text{with } flag \in \{comp, up, ups\}
\end{aligned}$$

Figure 4: Conversion from six-tuple model to delete-insert model

representation instead and we will therefore refer to it as six-tuple delta model or six-tuple model for short.

While the six-tuple model allows for a natural representation of partial deltas, it is more complex to handle six distinct sets during update propagation. We experienced that delta rules become rather complex. In particular, the join delta rules require a large number of joins to capture the interactions between the different delta sets.

Overly complex incremental expressions can be avoided by sticking to the delete-insert delta model for the update propagation. To do so, partial deltas need to be transferred to the delete-insert model. Furthermore, the result of the update propagation needs to be converted back into the six-tuple model. The general idea here is to extend the schema of the delta sets by adding a type flag column. This flag is used to indicate the type of individual delta tuples. Unknown attribute values (of partial deltas) are padded with NULL values. One could say that partial deltas are “encoded” as special kinds of complete deltas.

We proceed by describing the conversion from six-tuple deltas to delete-insert deltas, and continue with the conversion in opposite direction.

Six-tuple model to delete-insert model The equations for converting from the six-tuple model to the delete-insert model are given in Fig. 4. For a relation R it is straightforward to express the delta sets R_{ins} , $R_{un/uo}$, and R_{del} by means of the delete-insert model, because these sets are non-partial. To distinguish complete delta tuples from partial ones, they are assigned a type flag of value `comp`.

The remaining delta sets are treated as follows: Partial updates R_{up} and upserts R_{ups} are added to the insert set ΔR . Note that we need to distinguish them from “regular” insertions and updates, however. To this end, we add tuples with the same primary key value to the delete set ∇R . All other attribute values are padded with NULLs, because the initial attribute values of these tuples are unknown. Additionally we add a flag to indicate the type of the delta tuple. Note that the schema of ∇R is extended to accommodate the type flag. The partial deletions R_{delk} are also added to ∇R . Since R_{delk} contains primary key values only, the missing attributes are padded with NULLs. Note that a `up` flag is used for partial deletions. Partial deletions can however be distinguished from partial updates. While partial updates have a matching tuple in ΔR , partial deletions do not.

Example 2 Recall the running example introduced in Sec. 5.2. Fig. 3 depicts both, the old and the new state of the base relations *Cust* and *Addr*. Assume that a log-based CDC technique is used for *Cust* providing insertions, update pairs, and deletions. Further assume

ΔCust					ΔAddr		
CID	CName	CDiscount	CAddr		AID	ACity	ACountry
1	Adam	5%	1		1	Aachen	DE
2	Bob	0%	4		4	Dresden	DE
4	Dave	0%	4				

∇Cust					∇Addr			
CID	CName	CDiscount	CAddr	flag	AID	ACity	ACountry	flag
1	Adam	0%	1	comp	1	-	-	up
2	Bob	0%	2	comp				
3	Carl	0%	3	comp	3	-	-	up

Figure 5: Sample deltas converted to the delete-insert model

$$\begin{aligned}
R_{ins} &:= \Delta R \bar{\bowtie}_{pk} \nabla R \\
R_{un/uo} &:= \Delta R \bowtie_{pk} \sigma_{(flag=comp)} \nabla R \\
R_{up} &:= \Delta R \bowtie_{pk} \sigma_{(flag=up)} \nabla R \\
R_{ups} &:= \Delta R \bowtie_{pk} \sigma_{(flag=ups)} \nabla R \\
R_{del} &:= \nabla R \bar{\bowtie}_{pk} \sigma_{(flag=comp)} \Delta R \\
R_{delk} &:= \pi_{pk}(\nabla R \bar{\bowtie}_{pk} \sigma_{(flag \neq comp)} \Delta R)
\end{aligned}$$

Figure 6: Conversion from delete-insert model to six-tuple model

that change tracking is used for Addr providing insertions, partial updates, and partial deletions. The deltas converted to the delete-insert model are depicted in Fig. 5.

Delete-insert model to six-tuple model The equations for converting from the delete-insert model back to the six-tuple model are given in Fig. 6. Note that the symbols \bowtie and $\bar{\bowtie}$ are used to denote a semi join and a anti join, respectively. The R_{ins} delta set consists of those tuples in ΔR that have a primary key value not existent in ∇R . The $R_{un/uo}$ delta set consists of pairs of tuples in ΔR and ∇R having equal primary key values and being “complete”. The completeness is checked by means of the type flag in ∇R . The R_{up} and R_{ups} delta sets consist of tuples in ΔR that join to tuples in ∇R having a *up* or *ups* type flag, respectively. The R_{del} and R_{delk} delta set consist of tuples in ∇R having a primary key that does not exist in ΔR and being complete or incomplete, respectively.

5.4 Projection

The original delta rules for projection depicted in Tab. 1 are repeated for the reader’s convenience.

$$\begin{aligned}
\Delta(\pi_A(S)) &\equiv \pi_A(\Delta S) - \pi_A(S_{old}) \\
\nabla(\pi_A(S)) &\equiv \pi_A(\nabla S) - \pi_A(S_{new})
\end{aligned}$$

The projection delta rules contain a so-called effectiveness test to prevent redundant updates from being propagated. A set difference is used to discard insert delta tuples if an alternative derivation of the same tuple existed in the old database state. Similarly, delete delta tuples are discarded if an alternative derivation continues to exist in the new database state.

We can represent the new and old relation states S_{new} and S_{old} using the so called preserved state $S_o := S_{new} - \Delta S = S_{old} - \nabla S$, i.e. the set of tuples that have not been changed.

$$\begin{aligned}\Delta(\pi_A(S)) &\equiv \pi_A(\Delta S) - \pi_A(S_o \cup \nabla S) \\ \nabla(\pi_A(S)) &\equiv \pi_A(\nabla S) - \pi_A(S_o \cup \Delta S)\end{aligned}$$

Recall that dimension views contain primary key attributes that must not be dropped by a projection. Since primary key values are unique, $\pi_A(\Delta S)$ and $\pi_A(S_o)$ are obviously disjoint. Similarly $\pi_A(\nabla S)$ and $\pi_A(S_o)$ are disjoint. Given this, the delta rules can be simplified for dimension views as follows.

$$\begin{aligned}\Delta(\pi_A(S)) &\equiv \pi_A(\Delta S) - \pi_A(\nabla S) \\ \nabla(\pi_A(S)) &\equiv \pi_A(\nabla S) - \pi_A(\Delta S)\end{aligned}$$

The effectiveness test in the above equations can be understood as follows. An alternative derivation of a delta tuple must have the same primary key value. An alternative derivation of an insert delta tuple can thus only be found among the delete delta tuples and vice versa (recall that updates are represented as delete-insert pairs).

The aim of the effectiveness test is to discard so called ineffective updates, i.e. updates that do not change the view. In the presence of keys, an ineffective update occurs when all updated attributes are dropped by the projection. In this case, the initial state of the propagated attributes is equal to their current state. Thus, the update is ineffective w.r.t. the view.

We now discuss the implications of partial deltas w.r.t. the effectiveness test. In fact, the test may not work as expected here. The reason is that the initial state of an updated tuple may not be available. Hence, its effectiveness cannot be tested. Without having the initial state available, we do not know which attributes have been updated. Hence, we cannot know whether the update will affect the view². However, propagating ineffective updates is not problematic, because a view is not changed when an ineffective update is applied. While ineffective updates cause some overhead, the view does not become inconsistent.

Delta rules for projection can be generalized to handle partial deltas. To this end, the effectiveness test is only done for complete delta tuples and omitted for partial ones.

$$\begin{aligned}\Delta(\pi_A(S)) &\equiv \pi_A(\Delta S) - \pi_A(\sigma_{flag=comp} \nabla S) \\ \nabla(\pi_A(S)) &\equiv \pi_{A,flag}(\nabla S) - \pi_{A,comp}(\Delta S)\end{aligned}$$

²Note that our notion of ineffective updates is related to the notion of safe updates studied mainly in the context of integrity checking [Beh09]. Safe updates are an overestimation of true updates that can be computed more efficiently. For integrity checking, safe updates are often sufficient. The computation of safe updates has been proposed to improve efficiency. In contrast, our ineffective updates necessarily occur in the context of partial deltas.

$\Delta C_{ust}'$			$\nabla C_{ust}'$			
CID	CName	CAddr	CID	CName	CAddr	flag
2	Bob	4	2	Bob	2	comp
4	Dave	4	3	Carl	3	comp

Figure 7: Result of the sample incremental expressions $\Delta C_{ust}'$ and $\nabla C_{ust}'$

In the second delta rule, the schema of ΔS is extended by adding a type flag column which is assigned the value **comp**. Note that the effectiveness test may safely be omitted. Doing so may result in a larger number of ineffective updates being propagated. However, the rules become even simpler and may be evaluated more efficiently. Furthermore, note that the delta rules are closed under the model for partial deltas. That is, the result of the operation is again partial deltas.

Example 3 *Reconsider the running example. The first term of the dimension view definition D is $C_{ust}' := \pi_{CID, CName, CAddr}(C_{ust})$. We can apply the delta rules given above to derive incremental expressions $\Delta C_{ust}'$ and $\nabla C_{ust}'$.*

$$\begin{aligned}\Delta C_{ust}' &\equiv \pi_{CID, CName, CAddr}(\Delta C_{ust}) - \pi_{CID, CName, CAddr}(\sigma_{flag=comp}(\nabla C_{ust})) \\ \nabla C_{ust}' &\equiv \pi_{CID, CName, CAddr, flag}(\nabla C_{ust}) - \pi_{CID, CName, CAddr, comp}(\Delta C_{ust})\end{aligned}$$

The result deltas are depicted in Fig. 7. Note that the update to the customer tuple with ID 1 is ineffective and thus discarded.

5.5 Selection

The original delta rules for selection found in Tab. 1 are repeated here for the reader's convenience.

$$\Delta(\sigma_p(S)) \equiv \sigma_p(\Delta S) \quad \nabla(\sigma_p(S)) \equiv \sigma_p(\nabla S)$$

These equations need to be adapted to handle partial deltas. We will discuss each type of delta separately in the following.

The inserted tuples in S_{ins} are propagated if they satisfy the selection predicate and discarded otherwise. The deleted tuples in S_{del} are treated similarly. For the update pairs in $S_{un/uo}$ both, the initial and the current state have to be considered. If the initial and the current state satisfy the selection predicate the delta tuple is passed on as an update, i.e. it remains in $S_{un/uo}$. If neither the initial nor the current state satisfy the selection predicate the update pair is discarded. If the initial state did satisfy the predicate but the current state no longer does, the initial state is propagated as a deletion, i.e. it becomes part of S_{del} . Similarly, if the initial state did not satisfy the predicate but the current state does, the current state is propagated as an insertion, i.e. it becomes part of S_{ins} .

For partial updates S_{up} the initial state is not known. It could have either satisfied the selection predicate or not. Thus, given that the current state does satisfy the predicate, the

$\Delta Addr'$			$\nabla Addr'$			
AID	ACity	ACountry	AID	ACity	ACountry	flag
1	Aachen	DE	1	-	-	ups
4	Dresden	DE	3	-	-	ups

Figure 8: Result of the sample incremental expressions $\Delta Addr'$ and $\nabla Addr'$

resulting delta is either an insert or an update. Since we cannot distinguish these cases, the delta tuple becomes part of S_{ups} , i.e. the delta tuple changes its type and becomes an upsert. Given that the current state of an partial update does not satisfy the selection predicate, a deletion needs to be propagated. Since the initial state of the updated tuple is unavailable, a partial deletion (S_{delk}) is propagated. Note that this deletion may be ineffective, i.e. the tuple to be deleted may not be found in the view, because it did not satisfy the predicate in its initial state either.

The upsert delta set S_{ups} is handled in a very similar way as partial updates. Again, the initial state of delta tuples is unavailable. Given that the current delta tuple satisfies the selection predicate, it remains in S_{ups} . If it does not, it becomes part of S_{delk} . Again, the partial deletion may be ineffective.

Partial deletions contain primary key values only. Obviously, the selection predicate can generally not be checked without having the non-key attribute values. However, tuples in S_{delk} may safely be propagated in all cases. If a tuple with the same primary key value exists in the view, it is deleted. If no such tuple exists, the view remains unchanged, i.e. the deletion turns out to be ineffective. Ineffective deletions occur, when the original tuple did not satisfy the selection predicate and therefore never appeared in the view.

$$\Delta(\sigma_p(S)) \equiv \sigma_p(\Delta S) \quad \nabla(\sigma_p(S)) \equiv \sigma_{p \vee flag \neq comp}(\pi_{a,s(flag)} \nabla S)$$

$$\text{with } s(flag) := \begin{cases} \text{ups} & \text{if } flag = \text{up} \\ flag & \text{else} \end{cases}$$

Given these considerations, the original delta rules for selection can be adapted to partial change data. The rule to compute the insert set does not need to be changed. The rule to compute the delete set needs some adaptations though, because the selection predicate can only be checked for non-partial delta tuples (with $flag = comp$). All partial delta tuples are simply passed on. As mentioned before, partial updates may become upserts and the type flag needs to be changed accordingly. Note that the delta rules are closed under the model for partial deltas.

Example 4 *Reconsider the running example. The second term of the dimension view definition D is $Addr' := \sigma_{ACountry='DE'}(Addr)$. By applying the above delta rules the incremental expressions $\Delta Addr'$ and $\nabla Addr'$ can be derived.*

$$\Delta Addr' \equiv \sigma_{ACountry='DE'}(\Delta Addr)$$

$$\nabla Addr' \equiv \sigma_{(ACountry='DE') \vee flag \neq comp}(\nabla Addr)$$

The result deltas are depicted in Fig. 8. Note that partial updates are turned into upserts and effective partial deletions into possibly ineffective partial deletions.

		<i>S</i>					
		pre	ins	del	up	ups	delk
<i>T</i>	pre	-	ins	del	ups	ups	delk
	ins	ins	ins	-			
	del	del	-	del			
	un	un	ins	-			
	uo	uo	-	del			
	up	up	ins	-			
	ups	ups	ins	-			
	delk	delk	-	delk			

Figure 9: Join Matrix 1

5.6 Join

The original join delta rules found in Tab. 1 are repeated here for the readers convenience. Note that both, the new and the old state of the base relations are required to incrementally maintain join views.

$$\begin{aligned}\Delta(S \bowtie T) &\equiv (S_{new} \bowtie \Delta T) \cup (\Delta S \bowtie T_{new}) \\ \nabla(S \bowtie T) &\equiv (S_{old} \bowtie \nabla T) \cup (\nabla S \bowtie T_{old})\end{aligned}$$

In the DWH environment source systems are decoupled and base relations are usually available in their new state only. However, the old state can be reconstructed using the new state and the deltas. Given a relation R , the preserved state R_o can be computed by subtracting the insert delta set from the new state ($R_o := R_{new} - \Delta R$). The old state R_{old} can then be computed by adding the delete delta set to the preserved state. In the light of partial deltas, it may not be possible to fully reconstruct the old state, because delta tuples in ∇R may be partial. Recall that a flag is used to indicate the type of delta tuples in ∇R . We hence use a type flag in the old state R_{old} as well; the preserved tuples R_o are assigned with the distinct type flag **pre** ($R_{old} := \pi_{\dots, \text{pre}} R_o \cup \nabla R$).

In this paper, we focus on the maintenance of so called dimension views defined in Sec. 5.2. All join predicates used in dimension views follow a common pattern. They are equality predicates and involve the primary key attribute of at least one relation. In the following, we consider the join of two relations S and T with the join predicate $(S.a = T.pk)$ where $S.a$ is an arbitrary attribute of S and $T.pk$ the primary key attribute of T . It is important to understand that the type of the resulting (joined) delta tuples depend on the type of both input delta tuples. To adapt the join delta rules, all possible combinations of delta types need to be considered. The different combinations are represented by the matrices in Fig. 9 and Fig. 10. Consider the matrix in Fig. 9. The column headings represent the different delta sets of S participating in the join. From left to right, there are preserved tuples, insertions, deletions, partial updates, upserts, and partial deletions. For the sake of clarity, update pairs are shown in a separate matrix (Fig. 10). The row headings in the matrix represent the different delta sets of R participating in the join. The cells of the matrix indicate the delta type resulting from a join between the corresponding delta sets of S and R .

Consider the matrix cell at the intersection of the S_{ins} column and the T_{up} row, for in-

S	T_{new}	S	T_{old}	result
un	-	uo	-	-
un	-	uo	pre	del
un	-	uo	uo	del
un	-	uo	del	del
un	-	uo	delk	delk
un	-	uo	up	delk
un	-	uo	ups	delk
un	pre	uo	-	ins
un	pre	uo	pre	un/uo
un	pre	uo	uo	un/uo
un	pre	uo	del	un/uo
un	pre	uo	delk	up
un	pre	uo	up	up
un	pre	uo	ups	ups
un	ins	uo	-	ins
un	ins	uo	pre	un/uo
un	ins	uo	uo	un/uo
un	ins	uo	del	un/uo
un	ins	uo	delk	up
un	ins	uo	up	up
un	ins	uo	ups	ups

S	T_{new}	S	T_{old}	result
un	un	uo	-	ins
un	un	uo	pre	un/uo
un	un	uo	uo	un/uo
un	un	uo	del	un/uo
un	un	uo	delk	up
un	un	uo	up	up
un	un	uo	ups	ups
un	up	uo	-	ins
un	up	uo	pre	un/uo
un	up	uo	uo	un/uo
un	up	uo	del	un/uo
un	up	uo	delk	up
un	up	uo	up	up
un	up	uo	ups	ups
un	ups	uo	-	ins
un	ups	uo	pre	un/uo
un	ups	uo	uo	un/uo
un	ups	uo	del	un/uo
un	ups	uo	delk	up
un	ups	uo	up	up
un	ups	uo	ups	ups

Figure 10: Join Matrix 2

stance. The cell indicates that the join result of these delta sets is to be propagated as insertion. This is obvious considering that any tuple added to S has a key that is unique in S . Thus, the key cannot be in the view yet. Hence, the result of the join is an insertions w.r.t. the view.

Consider the three right-most columns in the matrix referring to partial updates, upserts, and partial deletions in S . These deltas lack certain attribute values. They hence cannot be joined to T_{old} , because the join predicate cannot be evaluated. Consider a partial update in S , for instance. Recall, that the initial state of the updated tuple is not available. Hence, it is unclear whether the updated tuple used to find a join partner in T before the update. The joined tuple is either an update w.r.t. the view (if it used to find a join partner) or an insertion (if it did not). Since these cases cannot be distinguished for partial updates, an upsert has to be propagated.

Upserts in S are propagated as upserts and partial deletions as (possibly ineffective) partial deletions. Note that partial S deltas are handled for joins in a similar way than partial deltas are handled for selections (see Sec. 5.5). The function s defined in Sec. 5.5 to translate type flags can thus be reused in the generalized delta rules for joins.

The matrix in Fig.10 represents joins involving update pairs in S . Recall that the new state of an updated tuple is joined to the new state of T (T_{new}) while the old state of an updated tuple is joined to the old state of T (T_{old}) in the delta rules for update propagation. The matrix shows all possible join combinations. Let s be an update pair in $S_{un/uo}$, s_{un} the new state of s , and s_{uo} the old state of s . The first two columns of the matrix indicate where s_{un} finds a join partner in T_{new} . There are the following possibilities: A join partner may not exist, it may be a preserved tuple, an inserted tuple, an updated tuple in its new

state, a partial update, or an upsert.

The third and fourth column indicate where S_{uo} finds a join partner in T_{old} . A join partner may not exist, it may be a preserved tuple, an updated tuple in its old state, a deleted tuple, a partial deletion, a partial update, or an upsert.

The fifth column indicates the type of the delta resulting from the joins. As an example, consider the second row of the matrix. It treats the case where s_{un} does not find a join partner in T_{new} , while s_{uo} used to join to a preserved tuple (i.e. the join attribute of s was updated). Hence, a tuple $S_{uo} \bowtie T_o$ used to be in the view and needs to be discarded now. Thus, the resulting delta is of type deletion.

The matrix in Fig. 10 reveals a pattern. Whenever s_{uo} joins to a complete tuple in T_{old} , namely a preserved tuple, an updated tuple, or a deleted tuple, the resulting delta tuple is again complete, i.e. an update pair or a deletion. When s_{uo} joins to a partial update or a partial deletion in T_{old} , the resulting delta tuple is either a partial update or a partial deletion. The distinction is made based on the existence of a corresponding delta tuple in the insert set $\Delta(S \bowtie T)$, when the deltas are converted to the six-tuple model (see Sec. 5.3). When s_{uo} joins to an upsert, the resulting delta tuple is either an upsert or a partial deletion. Based on these considerations and the considerations that lead to the first join matrix, a function j is defined to derive type flags for joined tuples from the type flags of the input tuples.

$$j(flag_s, flag_t) := \begin{cases} flag_t & \text{if } flag_s = \text{pre} \\ \text{comp} & \text{if } flag_s = \text{comp} \wedge (flag_t = \text{pre} \vee flag_t = \text{comp}) \\ \text{up} & \text{if } flag_s = \text{comp} \wedge flag_t = \text{up} \\ \text{ups} & \text{if } flag_s = \text{comp} \wedge flag_t = \text{ups} \end{cases}$$

The join delta rules are adapted as follows to handle partial deltas.

$$\begin{aligned} \Delta(S \bowtie T) &\equiv (S_{new} \bowtie \Delta T) \cup (\Delta S \bowtie T_{new}) \\ \nabla(S \bowtie T) &\equiv \pi_{\dots, j(S.flag, T.flag)}(S_{old} \bowtie \nabla T) \cup \\ &\quad \pi_{\dots, j(S.flag, T.flag)}(\nabla S \bowtie T_{old}) \cup \\ &\quad \pi_{\dots, s(flag)}(\sigma_{flag \neq \text{comp}}(\nabla S)) \end{aligned}$$

Once again, the delta rule for computing the insert delta set remains unchanged. The delta rule for the delete delta set is changed in two ways. First, a function j is used to derive the type of the resulting delta tuples from the type flags of joining tuples in S and T . Second, an additional term is added to the rule to handle partial tuples in ∇S . In this additional term the function s (defined in Sec. 5.5) is used to modify the type flag as needed. Note that the join delta rules propagate partial deltas as defined in Sec. 3. The join operation is thus closed under this model.

In Sec. 5.2 we have shown that views need to include primary key attributes to be maintainable using partial deltas. A simple join view includes the primary key attributes of both base relations. As we will see, not all of these key attributes are required to maintain dimension views though. In dimension view definitions, all join predicates have a common form. They are equality predicates involving the primary key of at least one base relation.

ΔD					∇D					
CID	CName	CAddr	ACity	ACountry	CID	CName	CAddr	ACity	ACountry	flag
1	Adam	1	Aachen	DE	1	Adam	1	-	-	ups
2	Bob	4	Dresden	DE	2	Bob	2	Berlin	DE	comp
4	Dave	4	Dresden	DE	3	Carl	3	-	-	ups

Figure 11: Result of the sample incremental expressions ΔD and ∇D

Reconsider the join of S and T with the join predicate $(S.a = T.pk)$ with $S.a$ being an arbitrary attribute of S and $S.pk$ and $T.pk$ being primary key attributes of S and T , respectively. Obviously $S.a$ is functionally dependent on its key $S.pk$. Thus, in the join view $T.pk$ is functionally dependent on $S.pk$. Thus each join view tuple is uniquely identified by $S.pk$ alone. Hence, partial updates, upserts, or partial deletions can be applied based on $S.pk$ only. In summary, dimension views remain maintainable when key attributes used in a join predicate are projected out hereafter.

Example 5 *Reconsider the running example. The sample dimension view was defined as $D := Cust' \bowtie_{(Cust'.Addr=Addr'.AID)} Addr'$. The incremental expressions ΔD and ∇D can be derived using the delta rules given above.*

$$\begin{aligned}
\Delta D &\equiv (Cust'_{new} \bowtie \Delta Addr') \cup (\Delta Cust' \bowtie Addr'_{new}) \\
\nabla D &\equiv \pi_{CID, CName, CAddr, ACity, ACountry, j}(Cust'.flag, Addr'.flag)(Cust'_{old} \bowtie \nabla Addr') \\
&\quad \cup \pi_{CID, CName, CAddr, ACity, ACountry, j}(Cust'.flag, Addr'.flag)(\nabla Cust' \bowtie Addr'_{old}) \\
&\quad \cup \pi_{CID, CName, CAddr, NULL, NULL, s(flag)}(\sigma_{flag \neq comp}(\nabla Cust'))
\end{aligned}$$

The result deltas are depicted in Fig. 11. When ΔD and ∇D are converted back to the six-tuple model, one obtains an insertion (ID 4), an upsert (ID 1), an update pair (ID 2), and a partial deletion (ID 3).

5.7 Putting it all together

In the previous sections, it has been shown that projection, selection, and join (with restricted join predicates) are closed under the model for partial deltas. Furthermore it has been shown that join views are maintainable if they include all non-functional dependent key attributes. Recall that dimension view definitions are assembled from these operations. We can thus conclude that dimension views are maintainable using partial deltas if all non-functional dependent key attributes are included.

6 Related work

View maintenance techniques have been adapted in several ways to deal with situations where input data is not or only partially available (cf. [GM95] for a survey). Work on

self-maintainable views aimed at maintaining a materialized view using just the deltas and the view itself, i.e. without accessing the base relations. *Partial-reference maintenance* considers only a subset of the base relations and the materialized view to be available. The *irrelevant update problem* means to decide whether an specific update leaves a view unchanged looking at the deltas and the view definition only, i.e. neither accessing the view nor the base relations. Interestingly, previous work has not considered deltas to be partial themselves, as we did here. This is probably because change capture is much less of a problem in the non-distributed environment.

Previous work on view maintenance in a *warehousing environment* [ZGMHW95, ZGMW98, AASY97, AAM⁺02] was focused on synchronization issues arising when base relations and materialized views reside on distributed systems. So called maintenance anomalies may occur when base relations are updated while view maintenance is performed concurrently. To prevent maintenance anomalies the Eager Compensating Algorithm (ECA) [ZGMHW95], the Strobe family of algorithms [ZGMW98], and the SWEEP algorithm [AASY97, AAM⁺02] have been proposed for SPJ views.

Preventing maintenance anomalies is an orthogonal problem to our work. ECA, Strobe, and Sweep make use of standard rules of algebraic differencing and tacitly assume non-partial deltas to be available. However, as we have seen, many CDC techniques used for warehousing provide incomplete deltas only. We believe that both, synchronization and handling of partial deltas are very relevant in the DWH environment. Thus, we feel our work is complementary.

We investigated view maintenance in the context of partial deltas in earlier work of ours [JD08, JD09]. In our previous work, we analyzed the impact of partial source deltas on view maintenance to understand which delta types (insert, update, delete, upsert) can still be reliably propagated. Such an analysis could, for example, reveal that a given view is maintainable w.r.t. insertions but not w.r.t. deletions for source deltas of a certain completeness. In our current work we took a different approach. We identified a restricted class of views (dimension views) that can be fully maintained using partial source deltas of any kind. We furthermore proposed a generalized algorithm for maintaining dimension views using partial deltas.

7 Conclusion

Maintenance of materialized views is an established research topic. More recently it has been proposed to use view maintenance techniques in the DWH environment where base relations and materialized views reside on different machines. However, previous work tacitly presumed that deltas captured at the sources are “complete”. We analyzed existing change capture modules and discovered that this assumption does often not hold in practice. In fact, change capture techniques may be unable to provide complete deltas or may provide partial deltas more efficiently. Thus, conventional maintenance techniques cannot be used in common DWH environments.

In this paper we studied view maintenance using partial deltas. At first, we introduced a

formal model for partial deltas. As we have shown, in general views cannot be maintained using partial deltas but there is a class of view that is maintainable. We referred to this class as dimension views, because of their close relation to dimension tables, which are typically used in star schemas. Based on our formal model for partial deltas, we then proposed a new view maintenance algorithm. To our knowledge, our algorithm is the first that allows for maintaining (a class of) views using partial deltas.

References

- [AAM⁺02] Divyakant Agrawal, Amr El Abbadi, Achour Mostéfaoui, Michel Raynal, and Matthieu Roy. The Lord of the Rings: Efficient Maintenance of Views at Data Warehouses. In *DISC*, pages 33–47, 2002.
- [AASY97] Divyakant Agrawal, Amr El Abbadi, Ambuj K. Singh, and Tolga Yurek. Efficient View Maintenance at Data Warehouses. In *SIGMOD Conference*, pages 417–427, 1997.
- [Beh09] Andreas Behrend. A Classification Scheme for Update Propagation Methods in Deductive Databases. In *International Workshop on Logic in Databases (LID)*, 2009.
- [GLT97] Timothy Griffin, Leonid Libkin, and Howard Trickey. An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.*, 9(3):508–511, 1997.
- [GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.
- [JD08] Thomas Jörg and Stefan Dessloch. Towards generating ETL processes for incremental loading. In *IDEAS*, pages 101–110, 2008.
- [JD09] Thomas Jörg and Stefan Dessloch. Formalizing ETL Jobs for Incremental Loading of Data Warehouses. In *BTW*, pages 327–346, 2009.
- [KC04] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. John Wiley & Sons, Inc., 2004.
- [KP81] Shaye Koenig and Robert Paige. A Transformational Framework for the Automatic Control of Derived Data. In *VLDB*, pages 306–318, 1981.
- [KR02] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., 2002.
- [Mic] Microsoft. *SQL Server 2008*. <http://www.microsoft.com/sqlserver/2008/>.
- [QW91] Xiaolei Qian and Gio Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.*, 3(3):337–341, 1991.
- [ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *SIGMOD Conference*, pages 316–327, 1995.
- [ZGMW98] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency Algorithms for Multi-Source Warehouse View Maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.

Cloudy Transactions: Cooperative XML Authoring on Amazon S3

Francis Gropengießer Stephan Baumann Kai-Uwe Sattler
Ilmenau University of Technology, Germany
first.last@tu-ilmenau.de

Abstract: Over the last few years cloud computing has received great attention in the research community. Customers are entitled to rent infrastructure, storage, and even software in form of services. This way they just have to pay for the actual use of these components or services. Cloud computing also comes with great opportunities for distributed design applications, which often require multiple users to work cooperatively on shared data. In order to enable cooperation, strict consistency is necessary. However, cloud storage services often provide only eventual consistency. In this paper, we propose a system that allows for strict consistent and cooperative XML authoring in distributed environments based on Amazon S3. Our solution makes use of local and distributed transactions, which are synchronized in an optimistic fashion, in order to ensure correctness. An important contribution of this paper is the evaluation of our system in a real deployment scenario upon Amazon S3. We show the strong impact of write operations to S3 on the transaction throughput. Furthermore, we show that fragmenting data increases write performance and reduces storage costs.

1 Introduction

Driven by the big players of the internet who want to better utilize their large data center infrastructures, cloud computing has often been viewed as the next paradigm shift or even big revolution in IT over the last few years. Normally the term cloud computing is understood as a mechanism to provide shared resources (computing capacity, software, data) on demand to multiple computers over the internet. Though the basic idea of centralization and hosting is not new, in fact it goes back to the mainframe era, cloud computing offers some new benefits and opportunities. First of all, availability and reachability guarantees given by the cloud provider by relying on highly redundant infrastructure simplifies building business-critical services. Second, the elasticity of resources allows to provide the illusion of infinite resources for customers to rent on demand. This is strongly related to the scalability: a customer can start with only a few machines and extend the number with a growing demand of resources. Together with a pay-per-use pricing model this helps to reduce the TCO dramatically and, therefore, is particularly interesting for small projects and companies that cannot afford large investments in infrastructure.

In cloud computing resources are provided at different levels, ranging from Infrastructure as a Service (IaaS) like Amazon Elastic Compute Cloud (short: EC2), where fundamental computing resources are rented, over Platform as a Service (PaaS) like Amazon Simple Storage Service (short: S3) or Google AppEngine, where customer-created applications can be deployed to the cloud, to Software as a Service (SaaS) like Salesforce or Google

Docs, where a provider's application already running in the cloud can be used over the Internet.

Apart from computing-intensive jobs, web and application hosting or scalable data analytics using MapReduce [DG04], storage and database services are an application class well-suited for running in the cloud. Examples like Dropbox, Ubuntu One as cloud file storage and Slideshare for sharing presentation slides already exist. These systems are all based on Amazon's S3, but also the deployment of full-fledged SQL databases using Amazon RDS or Microsoft SQL Azure is possible as well as using data management services as part of higher level SaaS solutions. Such database services are particularly promising for cooperative applications in which different users can share and edit a common set of documents and data. Google Docs or Microsoft Office Web Apps are well-known examples of these cooperative applications. Engineering design and media production processes could benefit from cloud-based database services in a similar way. In this work we consider media production using spatial sound systems based on the wave field synthesis [Ber88] as an example application. This technique enables the creation of more realistic surround sound for movies or concerts. During the design process the task of a sound designer is to animate static objects and to define their locations and movements as well as the characteristics of their surroundings. This way a listener can for example be given the impression to be in a cave or in a concert hall. The result of this design process is a scene description stored in an XML-based scene graph. However, apart from relying on a graph-based model and the usage of specialized authoring tools the approach we are going to present here is not limited to this application domain but rather usable for any kind of cooperative XML authoring.

In this context, cooperativeness means to allow multiple users to work at the same time on shared XML data and to exchange information in arbitrary directions without restrictions. This way it is possible for each user to adjust his own work to the current state of the project and the work of others. Furthermore, a cooperative authoring environment should support transactional semantics in order to ensure recoverability in case of transaction and system failures as well as strict consistency. The latter is needed to (i) guarantee that every user has the current state of the project and to (ii) prevent wrong design decisions of a user due to incorrect or outdated data.

Thus, the goal of our work is to build a cloud-based data management system for cooperative authoring of XML scene graphs which is as easily usable as a cloud storage system like Dropbox but supports transactional semantics. Building such a system on top of existing cloud platforms raises several questions. First of all, an appropriate storage abstraction and system have to be chosen, this can either be a low-level BLOB store like S3 or a full-fledged SQL database. Based on this decision a system architecture which maps a cooperative transaction model to the abstractions and operations of the underlying storage technology has to be designed. Amazon S3, for example, only supports eventual and read-after-write consistency (depending on the region where the service is provided) and atomic updates are restricted to single keys (tuples).

In this paper, we present such a system. Based on our previous work [GHS09], where we have developed a transaction model and an appropriate synchronization strategy for closely-coupled client/server-based workgroup environments, we discuss the design and

implementation of a cooperative transactional authoring system for XML data on top of Amazon S3 as an example. The contribution of our work is twofold:

- We discuss the implementation of distributed optimistic synchronization of XML updates using S3 as storage layer.
- We present results of our experimental evaluation using a real deployment which shows that fragmenting data reduces storage costs and increases write performance. Furthermore, we show that S3 write performance has a strong impact on the overall transaction throughput.

The remainder of this paper is structured as follows. Section 2 summarizes related work. In Section 3, we briefly introduce our data and fragmentation model, the operations on tree structured data and Amazon S3. Furthermore, we sketch our optimistic concurrency control protocol developed in our previous work. In Section 4 we describe our proposed system model which enables cooperative processing of XML data using Amazon S3. In Sections 5 and 6 we discuss the applied transaction model as well as synchronizing and committing transactions in more detail. Section 7 reveals how strict consistency upon an eventually consistent storage layer can be achieved. In Section 8 we consider some aspects concerning transaction and system recovery. Section 9 presents the results of our evaluation followed by a conclusion in Section 10.

2 Related Work

The CAP theorem [FGC⁺97] states that only two of the three properties – consistency, availability and network partitioning tolerance – can be fulfilled in a distributed environment. Most of the current solutions show a lack of consistency guarantees in favor of a higher availability. Examples are Amazon S3, Amazon SimpleDB, Dynamo [DHJ⁺07], Yahoo PNUTS [CRS⁺08] and Google Bigtable [CDG⁺08]. Hence, without further extensions, these systems are not suitable for cooperative environments.

Not only in cooperative environments, but also in fields like business or e-commerce a strong need for strict consistency exists. With Amazon RDS, Microsoft SQL Azure, and Google AppEngine three companies tried to fulfill the consistency requirements of their customers. These systems provide strict consistency and transaction support. In [KKL10] an evaluation regarding performance, scalability, and costs of these systems (amongst others) can be found. However, in these systems transaction processing is either restricted to a certain entity group (Google AppEngine) or it is only supported on a single database instance (Amazon RDS, SQL Azure). Since we assume distributed data, support for distributed transaction processing is essential. Due to this, these systems are not applicable to our use case without further extensions.

The endeavor of building databases upon cloud storage systems is not new. Our work is mainly inspired by [BFG⁺08], [DAA10a] and [DAA10b]. In [BFG⁺08] the design of a database system on S3 is described. The authors address in detail, how atomicity, consistency and durability of transactions can be fulfilled. Concerning isolation they argue that protocols, like the BOCC (backward-oriented concurrency control [KR79]) protocol, can only be partly implemented, as they need a global transaction counter, which might

become a bottleneck. For this reason we use timestamps for validation purposes, which are assigned by every transaction manager separately. This is possible, because the transaction managers run on virtual machines within an IaaS layer, where synchronized clocks can be assumed. We discuss this in more detail in Section 6.

The authors of [DAA10a] propose a scalable and elastic layered system approach, called ElasTraS, for transaction processing upon S3. They assume partitioned data as we do. In order to process transactions across different partitions they use minitransactions. Minitransactions were first used in Sinfonia [AMS⁺09]. They provide only very restricted transactional semantics. Precisely spoken, every data object accessed by a minitransaction must be specified before the minitransaction is started. This is almost impossible in design environments we consider. However, minitransactions can be used in order to implement optimistic concurrency control [AGS08].

In [DAA10b] the authors propose a transactional system approach for collaborative purposes, e.g., collaborative editing. They support transactions on so-called key groups, which can be established dynamically. Amongst others, the authors describe an implementation of their system on top of existing key-value stores, e.g., Bigtable. However, resolving the problem of how to ensure strict consistency on top of a weak consistent key-value store is left as future work.

With our endeavor of synchronizing transactions in distributed environments, we also intersect with current research projects in the field of distributed transactional memories. In these systems, a preferred solution for distributed transaction processing is to run transactions on a single site and move the accessed objects between different sites [HS07, ZR09]. Although, this is an interesting approach for future work, frequently moving fragments between different buckets might decrease the overall system performance.

Besides the approach of building a cooperative system upon an existing cloud storage layer, the development and hosting of a tailored cloud storage system is another possibility. Thereby, systems like Scalaris [SSR08] and Chubby [Bur06], which use the Paxos commit protocol [Lam02] to guarantee strict consistency, could be used as an entry point. Furthermore, separating transaction processing and data access, like proposed in [LFWZ09], is an interesting research area. It should be considered for future work.

3 Preliminaries

In order to understand the approaches proposed in this paper, we briefly sketch some basic concepts needed. We start with a characterization of the data model before going into details on the tree operations and concurrency control. A detailed description of all concepts can be found in [GHS09].

3.1 Data Model and Fragmentation Model

Basically, we assume XML data as a tree structure following the tailored DOM (short: taDOM) specification [HH03]. There, a tree consists of *nodes* with unique *node ids*, *node labels* and *node values*. Nodes are connected via *directed edges* denoting parent-child relationships. Figure 1 shows an example.

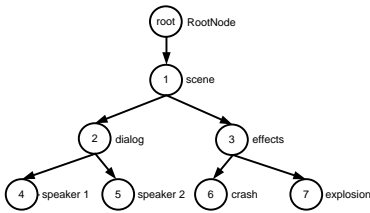


Figure 1: Example XML Tree

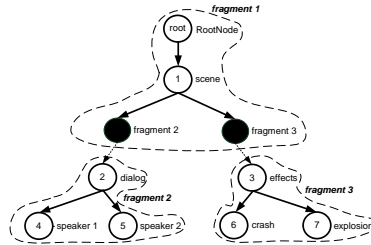


Figure 2: Fragmented XML Tree

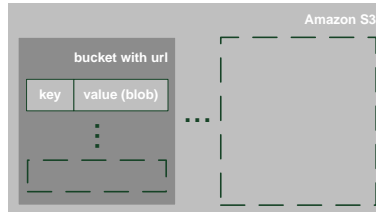


Figure 3: Amazon S3

Amazon S3 (see Figure 3), on the other hand, just provides a simple key-value store. Data is stored as blobs under unique keys. Furthermore, key-value pairs are organized in buckets, which are containers with a unique url. The task is to map our tree model to Amazon’s storage model. Therefore, we serialize tree data as simple strings and store these under unique names in S3 buckets. Serializing a whole tree results in a very large string. This leads to less read/write performance from/to S3 and hampers transaction throughput as we show later. Another possibility is to store every single node as a string using a unique key. However, this would increase costs, as multiple *get* operations have to be performed in order to retrieve partial tree data. Hence, a fragmentation model – as a trade off between costs and performance – is necessary. Figure 2 shows an example: The XML tree is shredded into three fragments with unique ids *fragment 1*, *fragment 2* and *fragment 3*. In order to reconstruct the whole tree, we introduce special nodes *fragment 2* and *fragment 3*, which reference the corresponding tree fragments. The reconstruction is performed in a top-down fashion. This approach is sufficient for our use case.

Fragmentation of a tree can be performed with respect to different aspects. Regarding our use case, sound production with spatial sound systems, a whole scene graph could be fragmented based on cooperation on data units. Assuming that in a sound studio different teams exist, e.g., one for effects and one for speech, then a possible fragmentation could be dialogs and effects as most in cases each team works on their own data set. However, there are other possibilities, for example fragmentation with respect to certain efficiency criteria. Summarizing, it depends on the application which fragmentation strategy should be used.

3.2 Operations

Our overall goal is to enable cooperative processing of tree data. In order to achieve this, we specified semantic tree operations. These allow for fine-grained conflict specification and hence support cooperative synchronization protocols.

In order to read a fragment and the contained tree, we defined a simple *read(fragment id)*. The manipulation of XML data is possible with the help of some update operations. An *edit(fragment id, n, new node value)* (E) is used to assign a new *node value* to a node *n*. The operation *insert(fragment id, n, sub tree)* (I) adds a new *sub tree* to node *n*. Thereby, a new edge between *n* and the root node of the *sub tree* is inserted. The operation *move(fragment ids, n, m)* (M) assigns node *m* as new *parent* of node *n* so that the existing edge (p, n) between *n*'s old parent *p* and *n* is removed and the new edge (m, n) is inserted. A *delete(fragment ids, n)* (D) removes node *n*, the edge (p, n) between *n* and its parent *p* and *n*'s children from the tree. Note that move and delete can affect several fragments.

Amazon S3 just offers really simple operations in order to read and update data, i.e., *get* and *put* for an object (blob).

The mapping of semantic tree operations to simple get and put operations is performed as follows. A fragment to be read is retrieved from S3 with the help of a get operation. Updates on the tree are performed externally with the help of semantic tree operations. In the end the changed fragment is written back to S3 using a put operation.

3.3 Synchronization Model

Simply using S3 operations in order to perform updates on data would have negative consequences regarding transaction synchronization, as a fine-grained conflict specification is impossible. Consider, for example, two authors who are working on the example scene graph in Figure 1. One author changes a dialog element and the other one changes an effects element. Both operations do not influence each other, because they are executed on different parts of the scene. However, both operations result in one put operation of the whole scene graph each and, hence, have to be considered as conflicting. Using semantic tree operations allows fine-grained conflict specifications. Furthermore, we consider functional dependencies between read and update operations. This leads to the following conflict definition:

Definition 3.1. *Two operations o_i and p_j , which belong to transactions t_i and t_j respectively, are conflicting, iff they are incompatible according to the compatibility matrix shown in Table 1. If both are *read(fragment id)* operations, they are not conflicting. Without loss of generality, we assume that o_i is a *read(fragment id)* operation and p_j is an update operation. Then both operations are conflicting, only iff o_i is followed by an update operation u_i which itself depends on o_i . Otherwise, they are not conflicting.*

Table 1 shows the compatibility of the update operations with respect to the nodes or edges our operations consider. Thereby, \checkmark states that the operations are fully compatible, $-$ that they are not compatible at all, and $+$ that they are only compatible if the tree they are performed on is considered unordered. Compatible operations are not in conflict and can

be executed in parallel. Read operations are not further considered. In our use case it is sufficient to investigate update operations in order to detect conflicts between different transactions.

	E	I	M	D
E	—	✓	✓	—
I	✓	+	+	—
M	✓	+	+	—
D	—	—	—	—

Table 1: Compatibility of Update Operations

In order to synchronize transactions, several approaches exist, each depending on the considered scenario. Pessimistic protocols, e.g. locking, are useful for working environments with high conflict rates. Optimistic protocols are applied in situations where a lower conflict rate is assumed. In our scenario we assume that in most cases authors are working on different parts of the same scene. Furthermore, the semantic tree operations lead to a lower conflict probability, as shown in Table 1. Hence, we apply an optimistic synchronization protocol, which is based on the well-known BOCC protocol. Here, we only present a short summary, more detailed information can be found in [GS10]. Like the traditional BOCC, we divide a transaction into three phases – read, validate and persist. Within the read phase all operations are performed on local copies of the data. In the validation phase the transaction is checked against all successfully committed transactions that interleaved the considered transaction. After successful validation the changes are stored persistently. Like the inventors of the BOCC protocol we assume validate and persist phase as an indivisible unit.

The interesting part of the protocol is the validate-persist phase. First, we summarize the validation criterion, as this is the main point where our protocol differs from the traditional BOCC protocol. For this purpose we introduce the notion of *UpdateSets*. An $UpdateSet_{T_i}$ contains all update operations a transaction T_i has performed. Update operations may or may not conflict, as it is shown in Table 1. Let T_j be a transaction with transaction number (or timestamp) $tn(T_j)$. T_j is successfully validated, iff $\forall T_i, tn(T_i) < tn(T_j)$ one of the following conditions holds:

1. T_i has completed its validate-persist phase before T_j starts its read phase.
2. $UpdateSet_{T_i}$ is not in conflict with $UpdateSet_{T_j}$ according to Table 1.

The first case follows the traditional BOCC approach. Hence, if T_i and T_j are executed serially, T_j is validated successfully. The second case implies that validation is not successful if the transactions conflict in their *UpdateSets*. Otherwise, if the transactions used different data items, serializability is preserved and T_j is validated successfully. The validation criterion works, because we assume less functional dependencies between read and update operations. In [GS10] we show that this validation criterion leads to fewer transaction aborts, even in case of environments with high conflict rates.

Finally, we briefly show correctness of our protocol.

Theorem 3.1. *The proposed validation criterion produces serializable schedules. Hence, every local transaction manager following this protocol produces locally serializable schedules.*

Proof. We construct a conflict graph G where nodes denote transactions and directed edges denote conflicts between these transactions [WV01]. A conflict is defined as in Definition 3.1. Iff G is acyclic, the schedule is serializable. Otherwise, it is not serializable. Now assume G is an acyclic graph of the form $(\dots \rightarrow t_i \rightarrow \dots \rightarrow t_k \rightarrow \dots)$. This implicitly means that all transactions were successfully validated and committed (Aborted transactions are not contained). By inserting a running transaction t_j this graph becomes cyclic. This means there must be at least one conflict directed from t_j to t_i and one directed from t_k to t_j , where $k = i$ is possible. However, if there are conflicts detected during the validation phase where t_j is involved in, then t_j is aborted. Hence, the graph stays acyclic. Note that validation is performed indivisibly. This means, only one transaction is validated at a certain time. Hence, if t_j is conflicting with another running transaction t_z , these conflicts are detected during validation of t_z (in case t_j has successfully been committed). \square

4 System Architecture

In Section 3 we discovered discrepancies between our requirements and what is provided by S3. Namely, these are the different data models and the different operation sets. Furthermore, we want to enable strict consistent and cooperative transaction processing upon S3 which by itself provides at most read-after-write consistency for put operations of new data. Hence, we need to define a system model which maps our data model and operations to S3 and provides strict consistency for the application.

Our proposed system architecture is shown in Figure 4. We chose a layered approach on top of Amazon S3.

Amazon S3 is organized in buckets, as described in the last section. In the context of media production with spatial sound systems we assume that all scene data (fragment sets) is partitioned over a set of buckets. This way every fragment belongs to exactly one bucket. The partitioning is performed in a way that a bucket contains the amount of scene data which is shared within a certain group of authors (clients). This is a natural approach with respect to huge movie projects, where we can find several teams for, e.g., sound effects or music.

Clients possess fragment caches in which they store local copies of the fragments they want to work on in form of trees. Every update operation is first executed on these copies. Only after the successful validation of the corresponding transactions the updates are stored persistently in Amazon S3. The advantage is that in case of a transaction abort no compensation (or version restoring) has to be performed in order to remove inconsistencies from S3. Clients are only allowed to read and update fragments via transaction managers within a transactional context.

The **VM layer** is situated between S3 and the clients. This layer consists of a set of virtual machines, which are started when needed and closed if they get dispensable. The virtual

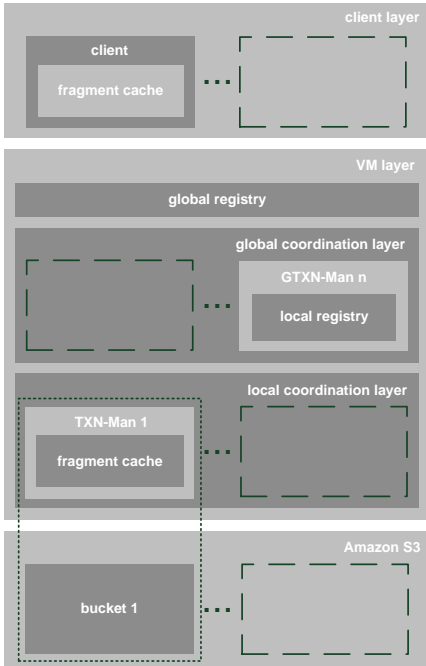


Figure 4: System Architecture

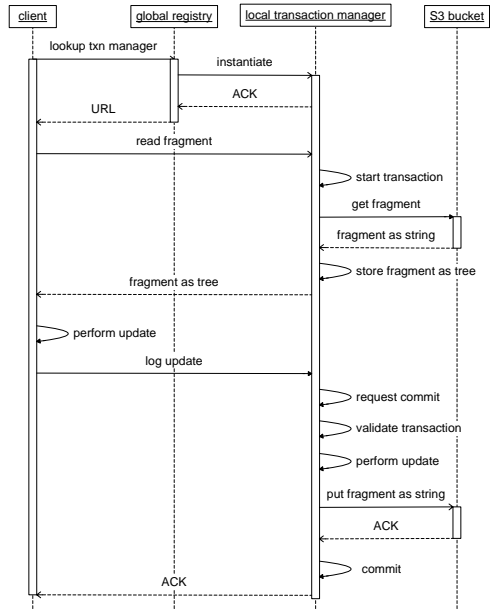


Figure 5: Sequence Diagram: Client Request

machines execute three kinds of services - local transaction managers (TXN-Man), organized in the local coordination layer, global transaction managers (GTXN-Man), organized in the global coordination layer and a single global registry. (Note that several services can run on a single virtual machine.) Every local transaction manager is exactly assigned to one single bucket and vice versa. It retrieves (get operation) copies of the scene data from the bucket it is assigned to, parses the XML string and stores this data in form of a tree in its fragment cache. The copies are written back to the bucket (in form of an XML string using the put operation) after manipulation. A local transaction manager is started/executed if at least one client intends to work on a certain bucket, otherwise it is stopped. Occasionally it is required that several groups of authors synchronize their work. This may result in concurrent access to different buckets. In order to handle this, global transaction managers are needed. Their task is to route client operations or requests to the responsible local transaction managers. However, they are not allowed to access S3 buckets. Hence, they have a local registry where they store which local transaction manager is responsible for which fragment of the tree. A global transaction executed by a global transaction manager is decomposed into several sub transactions. These are then executed by the responsible local transaction managers. Note that the sets of local transaction managers accessed by different global transaction managers do not necessarily have to be disjoint in order to allow for correct transaction synchronization. We show this in Section 6. Similar to local transaction managers, global transaction managers are only instantiated when needed. In addition to the transaction managers a single global registry is maintained. It is responsible for starting, stopping, and monitoring transaction managers and their corresponding

virtual machines. The registry knows which fragments are stored in which bucket. This is required in order to route client requests to the right transaction manager.

Next we give a brief example of how client requests are treated (Figure 5). Assume an author knows a fragment he wants to work on. It could have been assigned to him by the supervisor in his team. The client sends a lookup request to the global registry in order to get the address of the responsible transaction manager. In the case that an appropriate transaction manager is not yet running, it is instantiated. After obtaining the transaction manager's address, the author is able to start his work. Every atomic unit of work is encapsulated into a transaction. The client retrieves the fragment copy the author wants to work on with the help of a read operation. Thereby, a new transaction is started implicitly by the transaction manager. The transaction manager retrieves a copy of the fragment from S3 and stores it locally if it is not yet in the cache. Thereafter, the client (author) performs an update on the local copy which is also logged by the transaction manager. After a certain number of update operations were performed, the transaction manager tries to commit the transaction. Therefore, the transaction enters the indivisible validate-persist phase. If validation is successful, the logged updates are applied to the local fragment copy and the updated fragment is written back to the S3 bucket. The validation and committing of transactions is described in more detail in Section 6.

We briefly discuss two possible APIs which enable application developers to connect to the proposed system. Using the first approach, the client application directly communicates with the VM layer (see Figure 4). However, this way application programmers have to care about fragment caching and have to use the tree operations proposed in Section 3.2. Following the second approach, application programmers use a low-level client provided by us. This client software cares about fragment caching. Additionally, it maps application operations onto the tree operations used by our system. Hence, it would be possible to use, for example, XPath or XUpdate for XML processing on the application side. These operations are then transparently mapped onto semantic tree operations.

5 Transaction Model

In [GHS09] we defined a transaction model for cooperative XML processing. However, it has to be simplified for this use case, because of the following reasons. First, managing root transactions in such a distributed environment causes too much overhead. They are simply running too long. During this time a client could change the transaction manager and the transactional context has to be exchanged between the participating transaction managers. Second, splitting up operations on sub trees (like deleting a node with its corresponding children) into sub transactions (containing operations on nodes and edges) for recovery purposes is unnecessary, because S3 does not provide versioning of nodes and edges but only of fragments. Hence, a fine-grained recovery is impossible at all.

The resulting transaction model looks as follows. A transaction starts with a set of *read(fragment id)* operations followed by a set of update operations. Thereby, every data item that is affected by these update operations has to be read. This means we do not allow so-called "blind writes". Hence, we can assure that every author knows the current state of the project before he starts to make changes. This is necessary to allow for coopera-

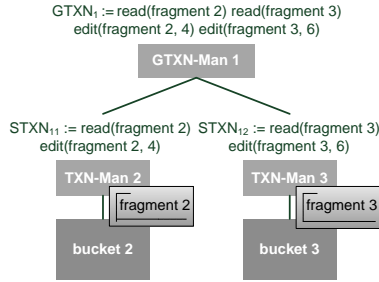


Figure 6: Example Transaction Execution

tive working. We illustrate our transaction model with a short example. Assume an author wants to balance the volumes of a crash and a speaker in the fragmented scene in Figure 2. Therefore, he reads fragment 2 and fragment 3 and performs two edit operations, one on node 4 and another on node 6. Figure 6 shows the resulting transactions in case that both fragments are stored in different buckets. In order to ensure atomicity across two buckets, a global transaction manager is used. All operations performed by the client are encapsulated in the global transaction GTN_1 . The global transaction manager knows the responsible local transaction managers and, hence, performs a decomposition of the global transaction into two sub transactions (STN_{11} and STN_{12}). In case both fragments are stored in the same bucket, no global transaction manager is needed. Then, the client communicates directly with the responsible local transaction manager and the example transaction is not decomposed at all.

Finally, the transaction boundaries (begin and commit) have to be determined. On one possibility they are explicitly specified by the designers or the application programmers. However, specifying transactions in design environments is not an easy task. In contrast to, for example, banking applications, where predefined transactions exist, e.g., for withdrawal, in design environments, the transactions are constructed during the interaction of a designer with the system. Usually, designers are not interested in specifying transactions. Hence, this task must be performed transparently by the system. In [GS10] we described an appropriate method, which automatically determines transaction boundaries based on a user defined degree of cooperation and the importance of the executed update operations.

6 Transaction Validation and Commit

In the last section, we described how it is determined, when a transaction is started and when it should be committed. Now, we clarify how transactions in this distributed environment are committed in order to guarantee atomicity and correctness.

In case an author works with XML data that belongs to a single bucket, there is no need for further investigation, because only a local transaction is executed at a single site. Here, validation is performed, and in case of success, the transaction is committed and the changes are stored persistently in S3. Correctness is guaranteed, because of our validation scheme described in 3.3.

However, in case an author works on scene data that is distributed across several buckets, a global transaction manager and several local transaction managers are involved. The

task is, *i*) to synchronize the global and local transactions against other global and local transactions in order to assure correctness and *ii*) to commit a global and all its local transactions entirely in order to guarantee atomicity. A common approach for these problems is the adaption of the two phase commit protocol [WV01] to optimistic concurrency control protocols. If a distributed transaction shall be committed, the global transaction manager becomes the *coordinator* and sends PREPARE messages to all affected local transaction managers. This message is a request for validation. Every local transaction is validated at its corresponding site. If validation is successful, a READY for commit message is sent to the coordinator. Otherwise, the local transaction manager answers with an ABORT message. If all local transactions voted for commit, the coordinator sends COMMIT messages to all participating local transaction managers and the whole distributed transaction is committed and the changes are stored persistently. If at least one transaction fails, all local transactions are aborted by the coordinator with the help of ABORT messages, followed by an abort of the global transaction itself.

Global transaction managers support parallel validation of several global transactions. However, local transaction managers treat the validate-persist phase including commit as indivisible unit, leading to serial validation.

The two phase commit protocol described above can lead to deadlocks between global transactions, because of cyclic wait-for graphs between sub transactions waiting for the entrance into the validate-persist phase. Two alternatives are known to deal with deadlock situations. First, avoid deadlocks - all transactions are validated simultaneously. If at least one sub transaction is not able to enter validation phase, the global transaction is aborted and, hence, all sub transactions are aborted, too. Second, deal with deadlocks - in the literature, e.g. [WV01], several mechanisms to detect and handle deadlocks are described.

In order to determine the serialization order of global and local transactions global transaction counters or timestamps are necessary. Since using a global transaction counter can become a bottleneck in highly distributed systems (as mentioned in Section 2), we chose timestamps for our solution. These can be assigned in a distributed manner. However, the precondition is, that clocks are synchronized at all transaction managers. In commonly known distributed systems this is hard to achieve. However, we assume our system (VM-layer) to be running in a data center on top of virtualization software like Eucalyptus, which also needs synchronized clocks. Current versions of the network time protocol provide time accuracy in the range of less than 10 milliseconds for local networks. We believe this is sufficient for our use case. However, for the unlikely case, that accuracy is too low, the approach described in [AGLM95] could be adapted and applied. However, this is beyond the scope of this paper.

Finally, we show that the proposed protocol is correct.

Theorem 6.1. *If every global transaction T_i follows the proposed commit protocol, the resulting schedule is globally serializable.*

Proof. Due to our proposed validation method, all locally executed transactions are serializable. Now, assume a schedule of committed global transactions T_k , $k \in \mathbb{N}$. We construct a global conflict graph S . A conflict is defined according to Definition 3.1. Iff S is acyclic,

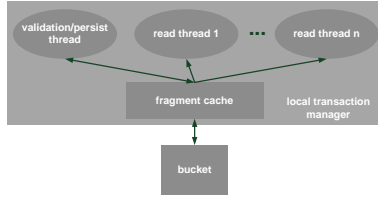


Figure 7: Local Transaction Manager Threads

the schedule is globally serializable. Assume, without loss of generality, that S contains a cycle of the form $(T_i \rightarrow \dots \rightarrow T_k \rightarrow \dots \rightarrow T_i)$. Hence, there must be a (direct or indirect) conflict directed from t_{ir} to t_{ks} and one directed from t_{ku} to t_{iv} , where t_{mn} is the n -th sub transaction of T_m executed at the n -th local transaction manager. Since all sub transactions were successfully validated and committed at the corresponding local transaction managers, T_i and T_k must have been serialized in opposite directions at different local transaction manager sites. This is not possible - at a local transaction manager site the validation phase is indivisible. Hence, only one transaction can be performed at a certain point in time. If a sub transaction entered the validation phase, it cannot leave it until its corresponding global transaction commits. (And a global transaction cannot commit if at least one sub transaction is not validated successfully.) This means, validation (serialization) order can only be unidirectional. \square

Note, that we did not assume, that the validation phase at the global transaction manager site must be indivisible. Hence, validation can be performed in parallel. This also means, it does not matter, if two global transactions are validated in parallel on the same or different global transaction managers.

7 Ensuring Consistency and Enhancing Read Performance

In order to understand how transaction throughput can be increased, we give some insights into the implementation of a local transaction manager. Figure 7 shows a local transaction manager with its corresponding S3 bucket. We implemented the transaction manager multi-threaded. We start exactly one validation/persist thread in order to serialize update operations and an arbitrary amount of reading threads, which are only allowed to read from the fragment cache. (In order to allow for real parallel reading, the number of read threads should follow the number of assigned CPU cores.) This implementation allows to serve a lot of client read requests in parallel.

The fragment cache is limited. Hence, it is possible that a cache resident fragment has to be replaced by another fragment that is immediately needed. In the literature, several page replacement strategies usable for our case are described. Thus, we do not consider fragment replacement any further. However, one problem remains with respect to fragment replacement. Assume a worst-case scenario where a fragment is updated and written to the corresponding S3 bucket. Right after this, it is replaced in cache by a different fragment and then immediately requested again and, thus, read from S3. As already said, Amazon S3 provides no strict consistency. So, if a fragment is written and read immediately thereafter, it is possible that an older version is retrieved. In order to prevent this, we make use of

the entire versioning feature S3 provides. An Amazon S3 put operation returns the version ID of the written object. We just store the returned version ID in the local transaction manager. This ID is updated every time a put of the considered fragment is executed. When the fragment is read, this version ID is passed to the Amazon S3 get operation. If the requested version is already available, it is returned. Otherwise, the transaction manager waits for some milliseconds and tries retrieving the fragment again.

8 Recovery

Concerning recovery we have to distinguish between transaction and system recovery.

Transaction recovery is necessary to guarantee atomicity in case of failures. Aborted transactions have to be rolled back to not have any effect on the persistent storage. In our approach transaction recovery and synchronization are closely coupled. In case of local transaction processing the use of our extended BOCC protocol ensures that only committed changes are stored persistently. Dirty updates are only performed on the client side during the read phase. Furthermore, cascading aborts are avoided as a transaction may only depend on previously committed transactions. (A transaction T depends on a transaction S if there is a directed conflict (Definition 3.1) from S to T .) For global transactions the variation of the two phase commit protocol ensures atomicity and durability. In cases where local or global transactions are aborted during read or validation phase, the affected transactions can just be aborted, as no changes have been stored persistently. Only the cases, where local transactions are aborted during the persist phase are a bit more intricate. However, Amazon S3 provides versioning feature, which is helpful in these situations. Even if a transaction is aborted after the put operation has been successfully performed, the previous version can easily be restored. Every put operation produces a new version of the fragment with a unique version ID. This version can simply be deleted by using the version ID. Since global transactions are not allowed to access S3 there is no need for further investigations.

A main problem of the original two phase commit protocol is that a deadlock situation occurs if the coordinator (global transaction manager) fails. Then, the local transaction managers are waiting for notifications infinitely. A possible solution is the introduction of time outs. In our system a global registry is running, which monitors all transaction managers. This way the registry can inform all affected local transaction managers if the global transaction manager fails and all local transactions are aborted. If a local transaction manager fails, the responsible global transaction manager is informed, which then reacts accordingly.

The base for ensuring system recovery is the global registry, which, as already mentioned, instantiates, monitors, and restarts virtual machines and transaction managers. In order to guarantee that the global registry is always available and also for load balancing purposes we use backup instances of this service. If a virtual machine or a transaction manager fails, it is detected by the global registry and the affected machines and managers are restarted. There is only a need for simple recovery steps after a local or global transaction manager was restarted. All running transactions have been aborted and all changes of committed transactions are stored in the corresponding S3 bucket. However, there could be

uncommitted changes in the bucket, as previously described. They can easily be removed by using the version IDs. After a local transaction was committed, the produced version ID is communicated to the global registry (during monitoring). If a local transaction manager is restarted, the last stored version ID is communicated to the local transaction manager, which simply retrieves a list of version IDs from the concerned bucket. If the last known version ID is not the current one in the list, the current version is simply deleted by using its version ID. Recovery for global transaction managers is much easier. A global transaction manager retrieves the lists of fragment IDs from its local transaction managers and can this way assure a proper routing of new client requests.

The versioning feature can also be used to allow undo operations. If a client wants to restore an old version of a fragment, it just retrieves all versions with a listing command via the global and/or local transaction manager. Then, it chooses one version and deletes all other versions. However, the undo feature has to be refined in order to enable for example access control. We consider this in future work.

9 Evaluation

In this section we evaluate the proposed system with respect to performance, applicability, scalability, and costs. We measure S3 performance and costs with respect to different fragment sizes in order to show that fragmentation reduces costs and enhances performance. Furthermore, we determine the local and global transaction throughput in order to show that the proposed system is applicable to cooperative media production scenarios. Concluding, we give a simple cost formula to help calculate the overall S3 costs and show that the proposed system is scalable within Amazon EC2.

In order to determine S3 costs we use prices for the region EU/Ireland from the Amazon S3 web page:

- *StorageCosts* = \$0.15 for storing 1 GB of data
- *GetCosts* = \$0.01 for 10000 get operations
- *PutCosts* = \$0.01 for 1000 put operations
- *DataInCosts* and *DataOutCosts* = \$0.15 for transferring 1 GB of data

The versioning feature is charged via storage costs. Every update operation leads to a full copy of the data item and hence increases storage costs.

Costs and Performance

As mentioned in Section 3.1, there are two possible extreme solutions for storing the tree data as blob objects – one blob per node or one blob for the whole tree. Here we outline how S3 costs and performance evolve between these two extrema and we show that fragmenting data reduces S3 costs and enhances data processing performance.

In order to measure the results we used an S3 bucket in the region EU/Ireland with the versioning feature enabled. Versioning is necessary for transaction recovery as described

in Section 8. The overall storage costs for a fragment of size $FragSize$ in MB, which is updated n times in an EU/Ireland bucket with enabled versioning feature, are calculated as follows: $TotalStorageCosts = FragSize * (StorageCosts/1000) * (n + 1)$. Furthermore, the whole get costs of retrieving all fragments of a tree are: $TotalGetCosts = GetCosts/10000 * NumberOfFragments$.

For our first measurement we consider a tree with a size of 500 KB. We fragment the tree step by step and consider get performance (the time needed for retrieving the whole data tree) and put performance (the time needed for putting a single fragment into the bucket). The tests were performed on an EC2 instance in region EU/Ireland as well as an instance in our institute. The results are shown in Figure 8. As expected, put performance increases with decreasing fragment size as decreasing fragment size avoids transferring of unchanged parts of the whole tree. The get performance decreases with an increasing degree of fragmentation. This can be explained by the higher number of get operations needed to retrieve the whole data tree and the associated higher number of expensive http requests sent to S3. Overall, results measured in EC2 are better than those measured externally. The reason is better infrastructure with less latencies and higher bandwidth.

Next, we measure $TotalStorageCosts$ and $TotalGetCosts$. Therefore, we, again, consider a tree with a size of 500 KB, which is stepwise fragmented. We assume that always the whole tree is retrieved from S3 and a single fragment of considered size is updated and written back to S3 ten times. The results are shown in Figure 9. As expected, the get costs increase and storage costs decrease with decreasing fragment size as decreasing fragment size avoids versioning of unchanged parts of the whole tree.

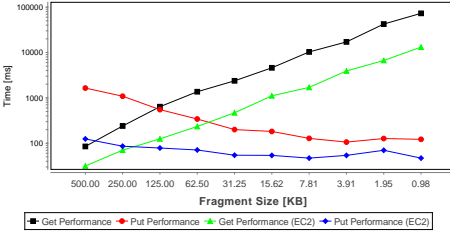


Figure 8: Put and Get Performance

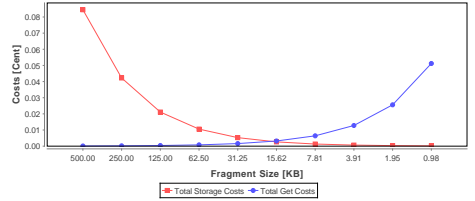


Figure 9: Total Storage Costs and Total Get Costs

Transaction Throughput

The goal of this part of the evaluation is to show that using the proposed system architecture and synchronization protocol results in sufficient transaction throughput. Furthermore, we show that fragmentation has some impact on transaction throughput.

Before presenting experimental results we first introduce the notion of *conflict rate*. The conflict rate determines the number of transactions which are conflicting with each other in the whole transaction set (for details see [GS10]). The set of conflicting transactions is determined based on the validation criterion of our new approach (conflicting *UpdateSets*). The tree the transactions are executed on is considered to be unordered.

We measure the number of successful committed transactions per minute ($TransactionThroughput = NumOfCommittedTxns/duration$, where $duration$ is the measured time in minutes for executing the whole transaction set) at the local/global transaction manager site. Thereby, we analyze the following scenarios:

1. Local transaction throughput depending on fragment size with fixed conflict rate of 25%.
2. Global transaction throughput depending on fragment size with fixed conflict rate of 25%.

For our test scenarios we used a simulation where a randomly generated set of transactions is executed on a randomly generated tree that follows our data model specification of Section 3.1. The tree is again fragmented step by step.

In test scenarios with local transactions the setup consisted of a local transaction manager upon an S3 bucket in the region EU/Ireland. In order to measure global transaction throughput the setup consisted of two local transaction managers upon two S3 buckets in the same region and one global transaction manager. All transaction managers are running on the same virtual machine. Local transactions consist of a set of read fragment operations and exactly one update operation. Global transactions consist of exactly two local transactions. It should be noted that in the local transaction scenario no global transactions are executed. Furthermore, in the global transaction scenario no local transactions that do not belong to global transactions are executed. Since duration of the read phase of a transaction is unpredictable (because it depends on the working time of an author), we assume a read phase duration between one and two seconds. Figures 10 and 11 show the measured results in case of transaction manager deployments in EC2 (region EU/Ireland) as well as in our institute. In both cases we measured transaction throughput when writing updates to

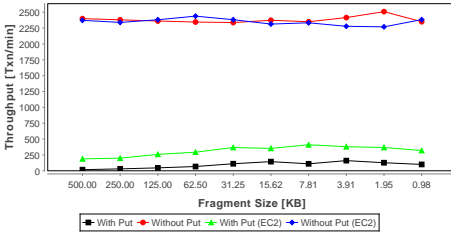


Figure 10: Local Transaction Throughput

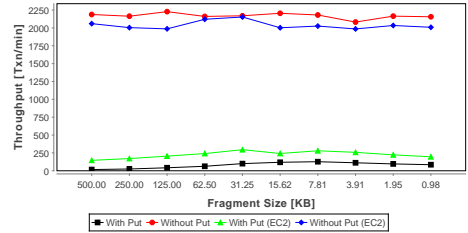


Figure 11: Global Transaction Throughput

S3 and in a second experiment only writing to the fragment cache. Basically, the measured transaction throughput is sufficient for typical cooperative design environments. However, write performance of S3, obviously, has a big impact on transaction throughput. In future work we have to think about sophisticated update caching mechanisms in order to save put operations. Furthermore, although writing a smaller fragment is faster than writing a large fragment to S3, transaction throughput slightly decreases with increasing fragmentation degree. We get this effect as with decreasing fragment size update operations affect

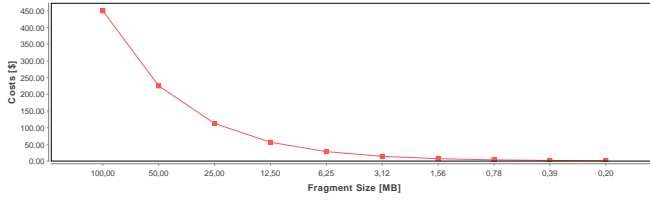


Figure 12: Total S3 Costs per Month

more fragments. Hence, more fragments have to be written back to S3 and this operation is heavily influenced by expensive http put requests.

Total S3 Costs

Here, we show that fragmentation reduces total S3 costs. We use the following cost formula in order to calculate costs resulting from external access to S3 (not from EC2):

$$TotalS3Costs = F * ((n + 1) * (FragSize * \frac{StorageCosts}{1000} + \frac{PutCosts}{1000} + FragSize * \frac{DataInCosts}{1000}) + n * (\frac{GetCosts}{10000} + FragSize * \frac{DataOutCosts}{1000})) \quad (1)$$

F is the number of fragments, n is the number of updates, and $FragSize$ is the size of a fragment in MB. Note that we assume that every update of a fragment leads to a new version of this fragment. Even if only a small part of the tree data within a fragment is changed, the whole fragment has to be read and written as one unit, because S3 is used as a block storage device.

These costs are worst-case costs, because we assume that get operations are always performed on S3. In real scenarios we assume most of the get operations on the cached fragments (local transaction manager). Hence, costs should be much lower.

Figure 12 shows the total S3 costs for one month, assuming a fixed project size of 100 MB and 10000 updates of the project file. The project file is fragmented step by step and updates are assumed to be distributed equally upon the fragments. This means, if we, for example, split up a single document into two fragments, $n/2$ updates are performed on each fragment. Again, we used a single S3 bucket in the region EU/Ireland with the versioning feature enabled. As expected, costs for the whole project decrease with an increasing degree of fragmentation. Hence, fragmentation should always be considered in big projects.

Scalability

Previously, we have shown that global and local transaction throughput is sufficient for typical design scenarios. For our measurements we used a relatively small test setup. Here, we want to show that our system model scales with an increasing workload. Therefore, we assume that the necessary transaction managers are hosted in EC2.

In EC2 it is possible to rent different instances with different memory and cpu equipment. For example, one can rent a machine with 7 GB RAM and 20 EC2 compute units. One EC2 compute unit compares to a single 2007 Opteron or Xeon cpu, which is at least a

dual core cpu. This means, we can execute at least 40 threads in parallel on one instance. As already mentioned in Section 4, it is possible to execute several transaction managers on a single virtual machine. Assuming a local transaction manager consists of one validate/persist thread and three read threads, we can run 10 local transaction managers (with corresponding S3 buckets) on a single instance. By assuming a local transaction throughput of 100 committed transactions per minute and local transaction manager, we get a total local transaction throughput of 1000 committed transactions per minute and instance.

If a bucket becomes a “hot spot”, we can split up the data across other buckets in order to distribute the load. If this is also not sufficient, fragments could be further decomposed and distributed across several buckets.

10 Conclusion and Future Work

In this paper, we proposed a system model, which enables strict consistent and cooperative processing of XML data in a distributed environment upon an existing cloud storage service, namely Amazon S3. We described applicable transaction models for local and distributed transaction processing, followed by a discussion of appropriate mechanisms for optimistic transaction synchronization and transaction commit protocols. We proved that these proposed protocols ensure correctness. Furthermore, we described how strict consistency can be achieved upon a storage layer that at most provides read-after-write consistency for put operations of new data. We also considered recovery with respect to transaction aborts and system failures. A critical part of this paper is the evaluation. We investigated the impact of S3 performance on our implementation within a real deployment on S3. We have shown that fragmentation is necessary in order to reduce storage costs and increase write performance. Furthermore, the evaluation reveals the strong impact of writing to S3 on the transaction throughput. For future work we propose to look for sophisticated approaches in order to avoid this bottleneck. A possible solution is caching of write operations. However, appropriate recovery mechanisms are necessary in order to guarantee durability.

References

- [AGLM95] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *In ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1995.
- [AGS08] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *PVLDB*, 1(1):598–609, 2008.
- [AMS⁺09] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3), 2009.
- [Ber88] A. J. Berkhout. A Holographic Approach to Acoustic Control. In *Journal Audio Eng. Soc.*, volume 36, pages 977–995. 1988.
- [BFG⁺08] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *SIGMOD Conference*, pages 251–264, 2008.
- [Bur06] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI*, pages 335–350, 2006.

- [CDG⁺08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), 2008.
- [CRS⁺08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!’s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.
- [DAA10a] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. *CoRR*, abs/1008.3751, 2010.
- [DAA10b] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.
- [DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI’04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10. USENIX Association, 2004.
- [DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon’s highly available key-value store. In *SOSP*, pages 205–220, 2007.
- [FGC⁺97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *SOSP*, pages 78–91, 1997.
- [GHS09] F. Gropengießer, K. Hose, and K. Sattler. An Extended Transaction Model for Cooperative Authoring of XML Data. *Computer Science - Research and Development*, 2009.
- [GS10] Francis Gropengießer and Kai-Uwe Sattler. Optimistic Synchronization of Cooperative XML Authoring Using Tunable Transaction Boundaries. In *DBKDA*, pages 35–40, 2010.
- [HH03] Michael Peter Haustein and Theo Härder. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. In *ADBIS*, pages 88–102, 2003.
- [HS07] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.
- [KKL10] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD Conference*, pages 579–590, 2010.
- [KR79] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. In *VLDB*, pages 351–351, 1979.
- [Lam02] Leslie Lamport. Paxos Made Simple, Fast, and Byzantine. In *OPODIS*, pages 7–9, 2002.
- [LFWZ09] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwillig. Unbundling Transaction Services in the Cloud. In *CIDR*, 2009.
- [SSR08] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Erlang Workshop*, pages 41–48, 2008.
- [WV01] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2001.
- [ZR09] Bo Zhang and Binoy Ravindran. Brief Announcement: Relay: A Cache-Coherence Protocol for Distributed Transactional Memory. In Tarek Abdelzaher, Michel Raynal, and Nicola Santoro, editors, *Principles of Distributed Systems*, volume 5923 of *Lecture Notes in Computer Science*, pages 48–53. Springer Berlin / Heidelberg, 2009.

Conceptual Views for Entity-Centric Search: Turning Data into Meaningful Concepts

Joachim Selke, Silviu Homoceanu, and Wolf-Tilo Balke

Institut für Informationssysteme
Technische Universität Braunschweig
{selke, silviu, balke}@ifis.cs.tu-bs.de

Abstract: The storage, management, and retrieval of entity data has always been among the core applications of database systems. However, since nowadays many people access entity collections over the Web (e.g., when searching for products, people, or events), there is a growing need for integrating unconventional types of data into these systems, most notably entity descriptions in unstructured textual form. Prime examples are product reviews, user ratings, tags, and images. While the storage of this data is well-supported by modern database technology, the means for querying it in semantically meaningful ways remain very limited. Consequently, in entity-centric search suffers from a growing semantic gap between the users' intended queries and the database's schema. In this paper, we introduce the notion of conceptual views, an innovative extension of traditional database views, which aim to uncover those query-relevant concepts that are primarily reflected by unstructured data. We focus on concepts that are vague in nature and cannot be easily extracted by existing technology (e.g., *business phone* and *romantic movie*). After discussing different types of concepts and conceptual queries, we present two case studies, which illustrate how meaningful conceptual information can automatically be extracted from existing data, thus enabling the effective handling of vague real-world query concepts.

1 Introduction

With the widespread use of the Web as primary information source, entity-centric search has become a common task for many people, with product search arguably being most prominent. In this context, typical entity types are mobile phones, movies, and books, but could of course also be people, news items or events. Although the handling of entity data traditionally falls into the domain of database systems [Che76], database methodology alone is becoming less and less adequate to master this task. Entities are no longer characterized by structured data alone but to a large extent also by semi-structured and unstructured information. For example, besides technical specifications, a typical e-shopping website features detailed textual product descriptions, expert reviews, and a large variety of user-generated content such as ratings, tags, and opinions. While modern database systems offer extensive technical capabilities for storing a large variety of data types (e.g., text documents, XML documents, and even multimedia content), the means for querying this data remain very limited [Wei07].

Therefore, recent research has been more and more focused on integrating information retrieval capabilities into database systems, in particular by structuring unstructured data for use by structured queries [WKRS09, CRS⁺07, MS06]. Most of this ongoing research focuses on extracting *precise* facts from textual data using methods from the area of information extraction [Moe06]. While preliminary results are promising, still many problems remain to be solved.

But to make things even more complicated, an analysis about product search we performed on the AOL search query log revealed the following: When searching for mobile phones, people very often include *vague* concepts (e.g. *business phone*, *portability*, or *for kids*) in their queries, about as often as they refer to precise technical product details (e.g. *weight*, *display diagonal size*, or *talk time*). Figure 1 illustrates the different types and respective frequencies of queries related to mobile phones we identified in the AOL search query log.

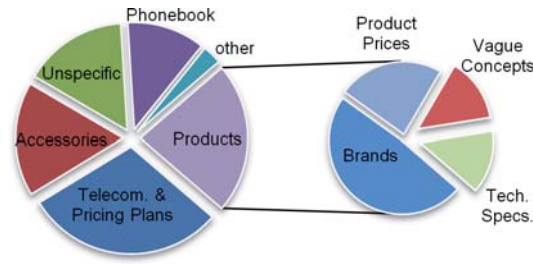


Figure 1: Different types of queries related to mobile phones in the AOL search query log

We also investigated what information about mobile phones is provided by current online shops, price comparison services, and media news sites such as CNET.com. We found that while almost all sites collect and allow searching for a broad range of technical specifications, the coverage of vague product features is fragmentary at best. Typically, information about concepts such as *business phone* is only available through manually-created top ten lists, which have been published as ordinary pages. User-defined top lists are particularly popular and even market leaders such as Amazon.com have recognized them as important means for providing conceptual information about products. None of the web sites we studied offered structured search functionality for vague concepts.

Our analyses indicate that there exists a significant mismatch between the users' intended queries and the database's schema. A very similar issue has been identified in the field of multimedia databases, where it is usually referred to as *semantic gap* [HLES06]. As argued above, in case of entity-centric search, the semantic gap mainly exists because users' information needs often are based on natural but typically vague concepts, which information providers usually do not model explicitly in their databases. However, previous studies also indicate that information about many query-relevant concepts is already contained in those parts of entity databases that are currently not used for answering the users' queries [SB10, KT09]. This mainly refers to unstructured information (e.g. textual product reviews), but may also include structured information (e.g. user ratings, which currently are mostly used to compute average product ratings, thus ignoring the users' hidden preferential structures).

In this paper, we present our approach to bridging the sematic gap in entity-centric search. As a key element, we introduce the notion of *conceptual views*, an innovative extension of traditional database views, which aims at systematically providing the means for making implicit conceptual information explicit to database applications. In particular, using case studies from the domains of mobile phones and movies, we demonstrate how conceptual views can be constructed automatically from the existing data and provide a rough classification of typical query types and matching extraction techniques.

The rest of the paper is structured as follows: First, we introduce and discuss the notion of conceptual views as well their use in modern database systems in Section 2. In the following sections we present our case studies. We continue by reviewing related work in Section 5, and conclude by highlighting some important findings from cognitive psychology in Section 6, which are strongly aligned to our approach and will guide our further work. We conclude by summarizing the results of our current research efforts and discuss open problems in Section 7.

2 Designing a View Mechanism for Answering Conceptual Queries

In this section we will discuss the basic mechanism for answering conceptual queries. Since database entities usually represent entities of the real world, the key idea is to understand concepts as special database attributes in a structured form. The attribute's value for every entity is obviously determined by the "degree of applicability" of the concept, which can be defined in a variety of ways as we will discuss later. In any case, this specific way of mediating between some user's or application's information need and the logical design of a database or information system is generally provided by the view mechanism. In the following we will briefly discuss how concepts are prepared for retrieval purposes using conceptual views.

2.1 Detecting Concepts in Queries

As we have seen, many queries address a rather conceptual understanding of database items (or entities) and therefore cannot be answered directly. But, how can such queries be handled in an effective yet easy-to-use way? The first step of course is to detect some new concept within queries, and thus a new information need. Whereas this is easy to do in SQL-style declarative query languages, where a mapping of previously unknown attributes to actually existing attributes in the underlying source(s) can be derived (see for instance the work on malleable schemas [ZGBN07]), the recognition of new concepts in simple keyword queries is somewhat harder. Of course, it is impossible to mine all individual concepts from a vast number of query terms put together in millions of queries regarding some topic. But preparing the underlying database to answer at least the most often occurring keyword queries, and thus providing for predominant groups of users, can be a strategic advantage, especially for e-commerce portals.

Following our running example, we therefore determined typical characteristics of predominant conceptual queries, i.e., what concepts with respect to mobile phones are there and how often do they occur? To answer this question we inspected the online advertising platform Google AdWords¹ and related the monthly number of general queries on our example domain of mobile phones (and all common spelling variants like “mobile phones” and “cell phones”) to the number of queries reflecting often used concept terms as derived from the AOL query log (again expanded with common spelling variants, but not related queries, e.g., the query “business phone” was not expanded by related terms like “calendar” or “organizer”). Since Google AdWords only allows for monthly averages, the results shown in Table 1 can only be seen as an intuition about the demand for individual concepts. For the month of September 2010, Google AdWords reported a total of 22,110,560 general purpose queries on mobile phones. Considering the top-5 concepts from the AOL query log we find that the relative monthly amount of queries ranges between 0.2% and 1.4%. Still, the result clearly shows that individual concepts will occur in significant numbers of queries and thus are easily detectable in a query log. Thus, periodically inspecting query logs for often co-occurring combinations of keywords can be expected to lead to the detection of currently relevant query concepts.

Concept	Frequency	
	Absolute	Relative
Cheap phone	313,400	1.4%
Business phone	87,520	0.4%
TV cell phone	62,700	0.3%
Music cell phone	49,300	0.2%
Cell phone for kids	33,260	0.2%

Table 1: Relative frequencies of concepts related to queries about mobile phones.

2.2 Building Conceptual Views

Since long relational databases have provided a mechanism for supporting queries that do not directly address attributes predefined in the logical design: views. Whereas views were often understood as a security feature regulating access to database tables and even providing some statistical data security by pre-aggregating several attributes, after the introduction of materialized views the performance implications became paramount. Especially for expensive aggregations a pre-computation and materialization of view attributes is essential. This perfectly fits to the complex nature of concepts and their problematic deduction from entity information.

The basic idea for building conceptual views is to derive each entity’s score with respect to some concept and offer it to query processing engines under the name of the concept (basically the used query term, for complex mappings of different queries to abstract concepts please refer to the large body of work on schema mapping). The score assigned

¹<http://adwords.google.com>

to each entity with respect to a concept can be interpreted in several ways. Of course the easiest way is to employ expert judgments simply rating all items. However, relying on editors (like e.g., the allmusic portal²) is an expensive and cumbersome method, which can only be employed on small collections, where trust in the scoring process is vital. On the other hand, given the variety of information about entities collected in today's databases and information systems, such as Amazon.com's shopping portal or the IMDb movie database³, conceptual information can be derived with adequate extractors. Before discussing these extractors, we will provide a brief overview of how concept scores are typically interpreted:

1. *Possibility that an entity represents a concept given structured information.* A good example is the concept *portability* for mobile phones or laptops. Here, the degree of membership (score) can be assumed to be a simple weighted aggregation of the weight and size attributes that will be part of the structured technical specifications.
2. *Possibility that an entity represents a concept, also considering unstructured information.* This is essential for concepts that cannot directly be derived from structured data, such as the concept of *business phone* in the domain of mobile phones. Usually, such concepts are to some degree based on opinions or user expectations, which are just supported by structured information.
3. *Probability that an arbitrary user would rate an entity as matching the concept.* The way of scoring is often modeled as degree of belief. A typical example is the notion of *beauty*, which again is sometimes supported by structured information, but in the end relies on (probably differing) opinions.
4. *Average user judgments.* User judgments already form a significant type of data in most information portals. Users are invited to express a personal opinion, and the scoring of each entity can then be derived by suitable aggregations of such ratings.

Of course, the major feat for the successful generation of conceptual views lies in the respective extraction algorithms for the concept scoring. Indeed, for the four interpretations above there are some typical extraction techniques (which we will describe in more detail and tied to conceptual query types in a later section). Generally speaking all extraction algorithms have to rely on a set of sample entities exhibiting the concept in question. Of course, such typical entities can always be provided by users in a query-by-example fashion (e.g., the *iPhone* as a typical smart phone or *Hugh Grant movies* as typical romantic comedies), but also a simple keyword search in unstructured data associated with some entities in our experiments proved to yield sufficiently accurate examples. The basic structure of extraction algorithms for the above interpretations can be roughly classified as follows:

1. *Extractors working only on structured data are usually of a purely statistical nature trying to find correlations between different attributes for the sample entities.* Generally attributes allowing for a good clustering of the sample, while showing a

²<http://www.allmusic.com>

³<http://www.imdb.com>

different overall distribution, can be expected to have some meaning with respect to the query concept. Typical algorithms like association rule mining for categorical data and Bayesian classification, or clustering algorithms for numerical data are well understood and already often used [WKQ⁺08].

2. *Extractors integrating structured and unstructured data usually involve some natural language processing techniques and are generally a mixture between statistical methods and techniques from information retrieval.* Since they form a currently very active and complex research topic, we will revisit a typical representative of these algorithms as use case in the Section 3 and discuss the result quality.
3. *Extractors for degrees of belief are usually relying on user relevance feedback in some form and thus tend to be interactive algorithms.* Generally speaking, all methods need some time to derive meaningful scorings, but following the wisdom-of-the-crowds principle [Sur04] eventually result in scorings of good quality. Due to their unobtrusiveness, recently the combined evaluation of query logs together with the results users clicked on has been a prime candidate for establishing degree of belief values [BHJ⁺10].
4. *Extractors for exploiting rating information often use an abstract semantic space for entity representations and then derive scorings from evaluating similarities in this space.* A typical representative of such algorithms is Latent Semantic Analysis [DDF⁺90]. Here, the feature space is rotated into the direction of prominent eigenvectors representing predominant topics that can be used to distinguish between sets of entities. We will also revisit this kind of extraction as a use case.

Having built a new attribute in the conceptual view for each relevant target concept using an adequate extractor depending on the type of concept, the view can be queried. Obviously, the extraction algorithms tend to be rather complex and time-consuming such that a materialized version of the view has to be maintained. This immediately raises questions about possibilities to update such views, which in turn reflects on the extraction algorithms used. However a detailed discussion of this problem is beyond the scope of this paper.

2.3 Answering Conceptual Queries

For answering conceptual queries by means of conceptual views, we must be aware of the dichotomy between precise concepts (which usually are already modeled explicitly in the database) and vague concepts (which are provided by conceptual views). The former typically will be used to specify hard logical constraints within the query (e.g., retrieve only Nokia phones or phones being cheaper than 300 Euros), while the latter are the primary focus of queries involving vague concepts (e.g., retrieve all *business phones* that are *mid-priced* and *iPhone-like*). Since those queries cannot be formulated and processed in a semantically meaningful manner using precise query languages such as SQL, a different approach is needed.

Since vague concepts almost always go hand-in-hand with the notion of degree of membership, a purely set-oriented retrieval approach seems inappropriate for conceptual attributes; ranking-based methods are more appropriate here. Fortunately, there already exists a large body of research dealing with exactly this type of query formulation and processing. As soon as all relevant concepts have been made explicit in structured form, a whole bunch of existing methods for supporting vagueness in queries and data can be applied, thus enabling a concept-enriched and more intuitive entity-centric search. Notable approaches are fuzzy databases [GUP06], the VAGUE system [Mot88], top-k retrieval [IBS08] and typicality queries [HPF⁺09], just to name a few. To integrate both types of query concepts, preference-based database retrieval [Kie02, Cho03] offers a large variety of options.

Figure 2 summarizes our vision of conceptual views and embeds this notion into the context of existing database systems. Conceptual views can actively and automatically be maintained by analyzing user query logs. As soon as relevant query concepts have been identified that currently cannot be handled using structured data, a suitable extractor is chosen to extend the conceptual view accordingly. Thus, conceptual views provide a systematic and unifying perspective on all available data, regardless of its type. Since all relevant concepts have been made explicit in structured form, existing methods for concept-based query processing can be applied to satisfy a broad range of information needs, which could not be handled using the previously existing structured data alone.

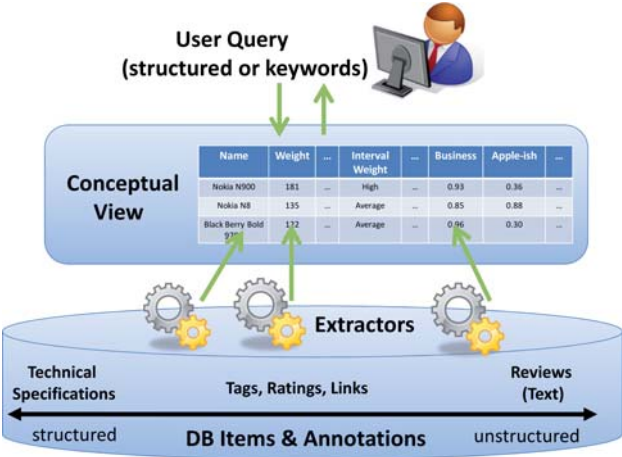


Figure 2: Conceptual views within a database system.

3 Case Study: Mobile Phones

Our first case study concerns the domain of mobile phones, which already has been discussed briefly. Here, in addition to providing a number of structured technical specifications for each phone, a typical product database also contains a textual description of the phone along with a (possibly large) collection of detailed reviews written by expert users or

journalists. The relevant query concepts vary from those that are primarily defined in terms of structured data (e.g., *portability*) to those that have almost no connection to the technical specifications (e.g. *well-designed*). In between, there are concepts being defined by both types of data (e.g. *business phone*). In this case study, we present a method that jointly analyzes both structured and textual data to extract a meaningful score value for some target concept, which in the following will be *business phone*.

In order to store the degree of membership for each entity towards the target concept, one first needs to build a model-based representation of this concept. Such a model comprises a feature collection together with the corresponding strengths, which we will refer to as *model vocabulary (MV)* in the following. Moreover, we will need an *entity representation function*, used for calculating the degree of membership of each entity in the database towards the target concept.

3.1 Method: Feature Analysis

The approach to be presented in the following is based on *conceptual features*, which are either “real” product features extracted from the technical specifications or nouns (and noun phrases) contained in some textual description or review of a product [LHC05]. We first extract all conceptual features from the available structured and unstructured data, and then try to find meaningful relationships between them (e.g., business phones tend to have advanced calendar functionalities). Our algorithm is based on a self-supervised learning technique that uses two types of training data for each product: a concept-related product review provided by some professional editor and the product’s technical specifications in structured form.

The method works as follows: We start by automatically splitting the training data (and thus also the entities) with classical information retrieval techniques (such as keyword search) into the explicitly concept-relevant data, further referred to as R , and the remaining data (for which the relevance towards the concept is unknown), further referred to as U . The sets R and U are disjoint. Of course, U will typically not only contain irrelevant entities, but also some entities for which the concept is only visible in terms of related features. In order to ensure a model of high quality, we have to split the training set by performing the concept keyword search both in the structured and unstructured data of each entity. We also have trained the model by using only editor product reviews which are extensive by nature, explicitly covering a broad spectrum of features and concepts.

Adapting procedures from document classification, we extract those product features that tend to discriminate entities in the set R from those in U (assuming that most entities in U will be irrelevant to the target concept). For this purpose, we assign a numerical strength to each feature, which measure the feature’s importance with respect to the given concept. We consider only the strongest ones for our MV. The strength of a feature f_i is defined as follows:

$$\text{strength}(f_i) = \frac{n_R(f_i) - \min_j(n_R(f_j))}{\max_j(n_R(f_j)) - \min_j(n_R(f_j))} - \frac{n_U(f_i) - \min_j(n_U(f_j))}{\max_j(n_U(f_j)) - \min_j(n_U(f_j))},$$

where $n_R(f_i)$ is the number of entities in R containing the feature f_i . The first summand calculates the normalized feature strength relative to entities belonging to R , while the second summand calculates the normalized strength relative to U .

Our method considers only those features having a reasonably high strength, namely three times the standard deviation above the average strength found in the entire population. An example of the resulting MV for the concept *business phone* is shown in Figure 3. The technical features and specification labels are extracted from the structured data, along with part of the corresponding values. Other features are extracted from unstructured data. Together with their corresponding strengths calculated with the above formula, all these features describe our target concept.

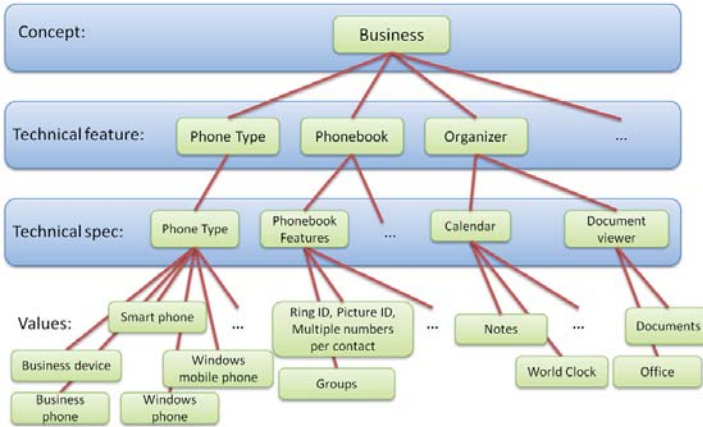


Figure 3: Model vocabulary for concept *business phone*.

In order to be able to evaluate the degree of membership of an entity E towards the target concept, we have used the entity representation function

$$\sum_{f_i \in MV \cap E} \text{strength}(f_i),$$

which states that an entity is as relevant to the concept as the sum the strengths of those features belonging to both the model as well as the entity.

As an example, consider that we want to compute the degree of membership towards the *business phone* concept for an entity which is described by the following text: “Powerful, but incredibly cumbersome. Pros: Has Microsoft Office, full featured calendar, support for multiple email accounts, internet connectivity, Wi-Fi, and a good battery life. Cons: It’s incredibly cumbersome and has a cluttered Windows 3.1 UI.” After evaluating the strength of this text only, by using our weighting function on the features from the text which also belong to the model (see Table 2), the entity gains a strength of 1.115, increasing its relevance towards the concept. Knowing that the total strength of the model is 57.082, the previous text provides for an increase in relevance by about 2%.

Feature	Strength
calendar	0.346
Wi-Fi	0.260
Windows	0.255
Office	0.155
battery life	0.099

Table 2: Features and associated strengths for concept *business phone*.

3.2 Experimental Setup

To evaluate our approach, we collected a training data set from PhoneArena.com, a major customer portal in the area of mobile phones. Our data set consists of expert reviews and technical specifications for 500 different phones. This data set has been used to build a model for the concept *business phone* as described above. Of course, this concept is not explicitly mentioned in PhoneArena.com’s structured data.

To test the predictive power of this model, we downloaded 200 user-provided reviews of the latest mobile phones from CNET.com. These reviews then have been manually labeled by experienced mobile phone users, either as being relevant or not relevant with respect to the concept *business phone*. We then compared the entity scores derived by our model to these manually created assessments. As evaluation metric, we used a precision–recall curve, which the dominant methodology for evaluating information retrieval systems. We compared our approach to two different baselines: document ranking by TF–IDF and Latent Semantic Indexing.

3.3 Results

The results of our evaluation are displayed in Figure 4. As we can see, our method is close to the two baseline approaches for high-precision scenarios, and outperforms them in high-recall settings. This clearly shows that integrating the available structured information into the retrieval process allows us to create a much more accurate model of the target concept than it is possible using previous methods. However, there is still room for improvement, which will be our primary goal in future work. To conclude, these results indicate that our method is well-suited for the task of constructing conceptual views from structured data and textual information.

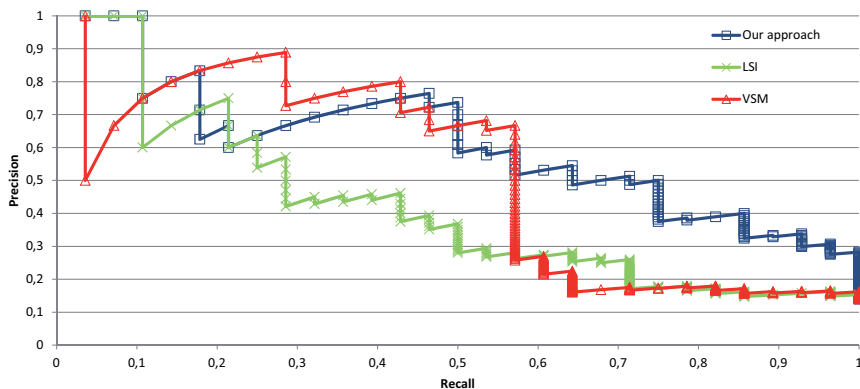


Figure 4: Precision–recall graph for the concept *business phone*.

4 Case Study: Movies

In our second case study, we are considering a database of movies. There are many popular examples on the web, e.g. IMDb, Netflix⁴, and Rotten Tomatoes⁵. Typically, those services offer their customers a broad spectrum of structured information about each movie, such as its title, release date, director, cast, genre, running time, and a short plot description. Also, they often allow people to contribute by providing their personal opinions in form of textual user reviews or ratings on a fixed numerical scale (e.g. one to five stars). The former usually are published on the respective service’s web site, the latter are used to compute a mean rating (which then is published) or to generate personalized movie recommendations for each user.

In contrast to our first case study, taste in movies is extremely complex and individual, and can only approximated very coarsely by the usual ways of cataloging movies. Therefore, the structured data available often is of very limited use for finding movies matching a user’s current mood or taste. To counter this problem, some providers have started to manually classify each movie along a wide range of semantically more meaningful concepts (e.g., *complexity* or *character depth*). This method is sometimes referred to as the Movie Genome approach and adopted by Clerkdogs⁶ and Jinni⁷, amongst others. However, since movie databases tend to be very large (ranging from around 10,000 movies in smaller systems to almost 1.7 million in larger ones such as IMDb), manually evaluating each movie with respect to many different vaguely defined concepts seems to be a challenging, if not impossible, task.

In the following, we demonstrate how such movie concepts can be made explicit by a conceptual view that extracts all necessary conceptual information from a large number of user ratings (where each user just assigns a number to each rated movie but does not

⁴<http://www.netflix.com>

⁵<http://www.rottentomatoes.com>

⁶<http://www.clerkdogs.com>

⁷<http://www.jinni.com>

provide any additional details). Each concept included in the conceptual view is defined by providing a small number of exemplary movie–score pairs. In a sense, this setting is similar to the machine learning task of semi-supervised learning [ZG09]. The approach to be presented in the following is based on but significantly extends previous work, which has been published recently [SB10].

4.1 Method: Semantic Spaces

In contrast to the feature-based approach presented in the previous section, we now purely rely on similarities and differences in users’ perception of movies, which are modeled by embedding the movies into an artificial high-dimensional coordinate space (“semantic space”). The individual dimensions of this space do not necessarily correspond to conceptual features of movies as recognized by humans.

We start by giving a formal definition of the problem to be solved. In the following, we use the variable m to identify movies, whereas u denote users. We are given a set of n_M movies and n_U users, where each user may rate each movie on some predefined numerical scale (e.g., the set of integers from one to ten). The provided ratings thus can be represented as a rating matrix $R = (r_{m,u}) \in \{\mathbb{R} \cup \emptyset\}^{n_M \times n_U}$, where each entry corresponds to a possible rating and $r_{m,u} = \emptyset$ indicates that movie m has not been rated (yet) by user u . Typically, the total number of ratings provided is very small compared to the number of possible ratings $I \cdot U$, often lying in the range of 1–2%. We are also given a small set of n movie–score pairs $C = \{(m_1, s_1), \dots, (m_n, s_n)\}$, which correspond to a human evaluation of the target concept for a random selection of movies. Our task is to estimate the score of all remaining movies.

In line with methodology that recently has been successfully applied in the area of collaborative recommender systems [KBV09], we first perform a factorization of the rating matrix R into two smaller matrices $A = (a_{m,i}) \in \mathbb{R}^{n_M \times d}$ and $B = (b_{i,u}) \in \mathbb{R}^{d \times n_U}$ such that the product $A \cdot B$ closely approximates R on all entries that are different from \emptyset ; the constant d is chosen in advance and typically ranges between 50 and 200. The idea is similar to the Latent Semantic Indexing (LSI) approach used in information retrieval [DDF⁺90]: Reduce the n_U -dimensional movie space (each movie is described by a vector of user ratings) and the n_M -dimensional user space (each user is described by a vector of movie ratings) to its most significant d -dimensional subspace.

Formally, the matrices A and B can be defined as the solution of the following optimization problem:

$$\min_{A,B} \text{SSE}(R, A \cdot B) + \lambda \sum_{(m,u) \mid r_{m,u} \in \mathbb{R}} \sum_{i=1}^d (a_{m,i}^2 + b_{i,u}^2),$$

where the SSE (sum of squared errors) function is defined as

$$\text{SSE}(R, \hat{R}) = \sum_{(m,u) \mid r_{m,u} \in \mathbb{R}} (r_{m,u} - \hat{r}_{m,u})^2$$

and $\lambda \geq 0$ is a regularization constant used to avoid overfitting. Besides this specific formulation of the matrix decomposition problem, many other versions have been proposed [KBV09]. However, the one just presented is the most fundamental.

To arrive at a canonicalized solution exhibiting some desirable properties (orthogonal axes, axes weighted by importance), we apply a singular value decomposition to represent the product $A \cdot B$ in the form $U \cdot S \cdot V$, where $U \in \mathbb{R}^{n_M \times d}$ is a column-orthonormal matrix, $S \in \mathbb{R}^{d \times d}$ is a diagonal matrix, and $V \in \mathbb{R}^{d \times n_U}$ is a row-orthonormal matrix. By reordering rows and columns, S can be chosen such that its diagonal elements (the singular values) are ordered by increasing magnitude. To arrive at a data representation that distinguishes only between movies and users, we integrate the weight matrix S to equal parts into U and V . Therefore, we define $U' = US^{\frac{1}{2}}$ and $V' = S^{\frac{1}{2}}V$ to be our final coordinate representation.

Now, each row of U' corresponds to a movie, and each column of V' corresponds to a user, both being represented as points in some d -dimensional space. This representation of movies provides the basis for learning the target concept from the examples in the set C . Taken together, the n_M movie points can be interpreted as a *semantic space*, which captures the fundamental properties of each movie [SB10]. Now, the target concept can be learned using specialized algorithms from the fields of statistics and machine learning. For this case study, we decided to use kernel-based support vector regression [SS04].

4.2 Experimental Setup

This case study uses the MovieLens 10M data set⁸, which consists of about 10 million ratings collected by the online movie recommender service MovieLens⁹. After post-processing the original data (removing one non-existing movie and merging several duplicate movie entries), our new data set consists of 9,999,960 ratings of 10,674 movies provided by 69,878 users (thus, about 1.3% of all possible ratings have been observed). The ratings use a 10-point scale from 0.5 (worst) to 5 (best). Each user contributed at least 20 ratings. Movie coordinates have been extracted using a method based on gradient descent [RIK07]. The regularization parameter λ has been chosen by cross-validation such that the SSE is minimized on a randomly chosen test set. We arrived at a value of $\lambda = 0.04$.

As target concepts to be learned from examples, we decided to use the collection of 37 concepts that have been manually created by movie experts from Clerkdogs. For each movie, an expert selected a subset of the available concepts (probably the most relevant ones) and scored the movie with respect to each of these concepts on a 12-point scale (0 to 11). We retrieved a total number of 137,521 scores for the 13,287 movie entries in their database (thus, each movie has been evaluated with respect to 10.4 concepts on average). After mapping these movies to the MovieLens 10M data set (and removing 9 movie entries which have been duplicates), we identified 7,813 movies that are covered by both data sets. Since the extracted coordinates of movies with only a small number of ratings by

⁸<http://www.grouplens.org/node/73>

⁹<http://www.movielens.org>

MovieLens users tend to be unreliable, we restricted our experiments to those movies that received at least 100 ratings. Finally, we ended up with a collection of 5,283 movies.

To learn each of the 37 target concepts from a set of examples, we randomly selected a subset of all the movies that have been scored with respect to the respective concept and applied kernel-based support vector regression to estimate the scores of all remaining movies. We then compared the estimated scores to the correct ones and measured both Pearson correlation and Spearman rank correlation to measure the estimates' accuracy. All experiments have been performed using MATLAB in combination with the SVM^{light} package¹⁰ for kernel regression. After some initial experiments we found the Gaussian radial basis function kernel to be most useful. We chose the learning parameters $C = 10$, $\gamma = 0.1$, and $\varepsilon = 0.1$ as they seemed to generate results of high quality. We did not yet perform a systematic tuning of these parameters.

We tried training sets of different sizes, ranging from 1% of the scored movies up to 90%. Although our scenario clearly focused on small training sets (to enable an easy definition of concepts within the conceptual view), we also included larger training sets to see the effect of the number of training examples on overall performance. For each combination of target concept and training size, we performed 20 experimental runs on randomly selected training sets. All numbers reported in the following section are averages over these 20 runs. Depending on the training size, the whole learning and estimation process took between 0.2 and 202 seconds on a notebook computer with a 2.6 GHz Intel Core Duo CPU (we used only a single core) and 4 GB of RAM.

4.3 Results

The results of our experiments are listed in Table 3. Since there have been large differences in performances among the different concepts, we abstained from aggregating the results into a single performance score over all target concepts. The table only reports the Pearson correlation between our estimations and the correct scores, as we found Pearson correlation and Spearman rank correlation to be extremely similar in most cases.

The most notable result is that all Pearson correlations are positive, that is, it was always possible to learn the target concept correctly at least to a certain degree. While for some concepts we have been able to achieve a quite high accuracy (e.g., *character depth* and *suspense*), there also have been concepts which proved to be hard to learn (e.g., *slow pace* and *revenge*); we can only speculate that these concepts do not significantly influence human movie preferences and thus are not reflected in the user ratings, but leave this question open for further research. We can also observe that for most concepts we can obtain a correlation between 0.2 and 0.3, even for a very small number of training examples. It is also interesting to see that even with small training sizes we are able to come close to the performance achieved on extremely large sizes.

Given that the correlation coefficient of a perfect estimation is 1 and that of a naive baseline (estimating each score by the average score in the training set) is 0, the estimated scores

¹⁰http://www.cs.cornell.edu/People/tj/svm_light

Concept	#movies	Size of the training set						
		1%	2%	5%	10%	20%	50%	90%
Action	2480	0.35	0.43	0.50	0.53	0.55	0.59	0.62
Bad Taste	470	0.09	0.18	0.29	0.38	0.44	0.49	0.49
Black Humor	1102	0.09	0.15	0.25	0.28	0.30	0.36	0.36
Blood & Gore	787	0.21	0.27	0.35	0.43	0.47	0.52	0.54
Cerebral	468	0.22	0.25	0.40	0.43	0.45	0.47	0.47
Character Depth	5198	0.68	0.70	0.72	0.73	0.75	0.76	0.76
Cinematography	3019	0.35	0.41	0.46	0.49	0.51	0.54	0.56
Complexity	2611	0.43	0.48	0.52	0.54	0.56	0.59	0.61
Crude Humor	677	0.23	0.36	0.48	0.53	0.57	0.60	0.62
Disturbing	1804	0.30	0.37	0.45	0.49	0.51	0.54	0.56
Downbeat	2831	0.28	0.32	0.38	0.40	0.43	0.46	0.47
Dry Humor	1656	0.15	0.18	0.26	0.29	0.32	0.37	0.40
Fantasy	604	0.10	0.13	0.20	0.27	0.32	0.41	0.48
Fast Pace	2988	0.12	0.15	0.19	0.22	0.23	0.25	0.29
Geek Factor	636	0.15	0.24	0.33	0.40	0.46	0.51	0.59
Hollywood Feel	2885	0.16	0.22	0.28	0.32	0.34	0.38	0.41
Humor	800	0.11	0.20	0.28	0.41	0.48	0.55	0.56
Informative	168	0.09	0.06	0.14	0.18	0.24	0.26	0.26
Offbeat	2414	0.33	0.37	0.40	0.43	0.45	0.47	0.51
Parental Appeal	521	0.20	0.37	0.45	0.48	0.52	0.54	0.58
Political	400	0.15	0.15	0.15	0.19	0.25	0.31	0.32
Revenge	526	0.08	0.09	0.14	0.20	0.25	0.29	0.29
Romance	2728	0.22	0.28	0.37	0.41	0.44	0.48	0.51
Screwball Humor	1133	0.13	0.18	0.23	0.27	0.29	0.34	0.34
Sex	2122	0.23	0.30	0.38	0.43	0.48	0.52	0.53
Slapstick Humor	1098	0.14	0.18	0.26	0.34	0.35	0.40	0.44
Slow Pace	2111	0.09	0.14	0.20	0.23	0.22	0.25	0.27
So Bad It's Good	154	0.02	0.04	0.03	0.09	0.15	0.22	0.27
Soundtrack	1932	0.19	0.24	0.31	0.35	0.39	0.44	0.47
Special Effects	655	0.20	0.24	0.30	0.36	0.42	0.46	0.50
Suspense	2693	0.36	0.44	0.51	0.54	0.57	0.60	0.62
Tearjerker	520	0.01	0.03	0.10	0.14	0.20	0.23	0.25
Terror	678	0.15	0.22	0.35	0.42	0.50	0.54	0.52
Truthfulness	177	0.14	0.29	0.35	0.42	0.47	0.49	0.49
Upbeat	2448	0.18	0.26	0.33	0.38	0.41	0.44	0.44
Violence	2914	0.33	0.42	0.49	0.52	0.55	0.60	0.63

Table 3: Pearson correlations for different target concepts and numbers of training examples.

do not seem to be very accurate. But this assessment takes too narrow a view, as it relies on the assumption that the scores provided by Clerkdogs' experts are indeed objectively correct. In our analysis of Clerkdogs' data we found nine movies that occur twice in the movie database, and thus have also been evaluated more than once by the experts, most probably without being aware of it. In total we located 63 movie–concept combinations which have been assessed twice, and used this data to estimate the inter-expert consistency. We found that the Pearson correlation between the rating pairs is only 0.60, which is an surprisingly low value. Therefore, the quality for most of our own results lies somewhere in the middle between a naive approach and a human expert assessment, which makes the results of this study a promising starting point for further research. Since our estimates are based on a broad range of user opinions, there is hope that at least for some concepts the wisdom of the crowd can outperform the experts [Sur04].

To conclude this case study, we have been able to show that a meaningful conceptual view can be created from rating data and a few examples of each target concept. Due to the short computation times, new concepts can be integrated easily.

5 Related Work

Our general approach and methods are related to two other prominent areas of research, namely, multimedia databases and recommender systems.

5.1 Multimedia Databases

The notion of semantic gap as used in this paper originates from research in content-based image and video retrieval, where the mismatch between the users' information needs and the available descriptive information has very early been identified as one of the foremost problems to be solved [HLES06]. Consequently, early approaches focused on extracting numerical scores from images and movies that are related to human perception such as the coarseness of an image, its color distribution, and the parameters of mathematical models describing textures, so called low-level features [SWS⁺00]. Although these features do not correspond directly to query-relevant concepts, they provide a solid foundation for further processing, in particular for defining meaningful similarity measures. In a sense, these feature spaces correspond to the coordinate spaces created by LSI and the method we presented in our second case study.

More recent research in multimedia databases, focuses on integrating high-level semantic features into the retrieval process [LZLM07]. Here, the idea is to learn relevant query concepts from the collection of extracted low-level features, thus bridging the semantic gap. This general approach follows the same spirit as conceptual views, although the latter also integrate meaningful structured data, which rarely exists in multimedia databases.

5.2 Recommender Systems

Recommender systems [AT05] aim at learning the user's preferences based on his or her previous interaction with the system. Typically, this is done by recording which items have been bought or at least investigated in the past. The main goal of recommender systems is to present a ranked list to the user containing those items which are likely to match the user's taste. Again, there is connection to conceptual views. While conceptual views try to extract concepts that are meaningful for all users, the only concepts relevant to recommender systems are of the general type *well-liked by user X*. Although the task of learning those concepts is much more focused than the extraction problem underlying conceptual views, it also limits the possibilities of recommender systems. For conceptual views, we intentionally chose not to perfectly fit any user's needs and taste but provide a semantically meaningful conceptual description of all entities. In particular, this enables the system to respond to spontaneous changes in the user's mood and taste (e.g., the lover of documentary movies that sometimes just wants to view a comedy). However, as we have seen in our second case study, research in recommender systems offers a wide variety of methods that can be adapted to construct conceptual views.

6 Cognitive Psychology's View on Concepts

Before concluding this paper, we would like to offer cognitive psychology's perspective on concepts. Until now, our motivation and handling of vague concepts has been backed mainly by intuition and common knowledge. While this approach is perfectly valid and in line with previous work on handling concept-related queries in database and information retrieval systems, we next show that many ideas presented are backed by research performed on cognitive psychology over the last forty years.

From a psychological perspective, concepts are mental representations of classes, and their primary function is to enable *cognitive economy* [Ros78]. By dividing the world into categories, we decrease the amount of information to be perceived, learned, remembered, communicated, and reasoned about. Concepts are considered to be formed through the discovery of correlations between features/attributes (that is, clear properties of the entities under consideration). For example, the concept *bird* is formed when noticing a correlation between the features *has wings* and *has feathers*. Features largely correspond to precise attributes that are typically modeled explicitly in databases.

When investigating how people think about concepts and categories, psychologists came to differentiate two kinds of categories: *precise concepts* (also called classical or crisp concepts) and *vague concepts* (also called fuzzy or probabilistic concepts). Precise concepts can be defined through logically combining defining features, e.g., the concept *prime number* can be defined this way. Vague concepts cannot be so easily defined, a popular example is *game*. Their borders tend to be fuzzy. Some concepts may even appear both in a precise as well as in a vague shape. Biologists may suggest that we use the word *fruit* to describe any part of a plant that has seeds, pulp, and skin. Nevertheless, our natural, vague

concept of fruit usually does not easily extend to tomatoes, pumpkins, and cucumbers. The notion of vagueness as used in this context is clearly to be distinguished from the problem of missing knowledge. Vague concepts are inherently fuzzy, that is, even with perfect knowledge of the world they do not allow crisp classifications of all entities. Vague concepts are primarily based in human intuition and often cannot be made explicit in terms of logical rules.

Another central property of human concepts is typicality, that is, within a category, items differ reliably regarding their “goodness-of-example.” While both penguins and robins clearly are considered birds, the former are judged as being untypical instances of this concept. This phenomenon can even be observed in precise concepts. For example, people generally consider 13 to be a better example of a prime number than 89 [Mur04].

Cognitive psychology also offers formal models for capturing the essential properties of real-world concepts. These models can roughly be classified into two groups: models based on features, and models based on semantic spaces. In feature-based models, each entity is represented by a list of features; the process of categorization often is modeled by complex interactions between these features, e.g., by means of neural network models. Models based on semantic spaces represent each entity as a point in some high-dimensional space, where individual coordinate axes do not necessarily have an interpretable meaning; categorization is modeled by means of similarity measures in this space, e.g., distance of an entity to the concept’s prototype. Both types of models have been found to be highly accurate; some researchers even propose to create hybrid models to get the best of both worlds [RJ10].

Now, the connection to the models used in our case studies becomes apparent. The model used in the first study (mobile phones) models the target concept using a feature-based approach; classification is performed based on relative feature frequency. In our second study (movies), the underlying model uses a semantic space for categorization, which has been extracted from rating data. Therefore, our work provides a solid foundation for further research and may readily be extended to incorporate the important notion of typicality.

7 Conclusion and Outlook

In this work, we identified and discussed one of the major problems of entity-centric search, namely, the lack of support for querying natural concepts in a structured fashion. We demonstrated that most of the relevant information is already present in current database systems and only needs to be made visible to applications.

To this end, we proposed the notion of conceptual views, which provide a systematic and unified interface between the broad range of data types available in current database systems and existing query processing algorithms, thus bridging the semantic gap between the vague concepts characterizing the users’ information need and the database schema.

In two extensive case studies from different domains we have been able to demonstrate that our vision of an automated extraction process for vague concepts can indeed be put into practice. We also showed that our work is backed by theories from cognitive psychology.

However, this paper also revealed that we have just started our journey towards an effective, efficient, and reliable construction and maintenance of conceptual views. In future work, we will develop a detailed theory of conceptual views that is closely interwoven with models and theories from cognitive psychology. Since we are primarily dealing with natural, vague concepts, there needs to be a stronger focus on research results from the humanities. Moreover, we are going to both continue the work on our existing concept extractors as well as creating new ones for application scenarios that are not covered well enough yet.

References

- [AT05] Gediminas Adomavicius and Alexander Tuzhilin. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, 2005.
- [BHJ⁺10] Dominik Benz, Andreas Hotho, Robert Jäschke, Beate Krause, and Gerd Stumme. Query Logs as Folksonomies. *Datenbank-Spektrum*, 10(1):15–24, 2010.
- [Che76] Peter Pin-Shan Chen. The Entity-Relationship model—Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
- [Cho03] Jan Chomicki. Preference Formulas in Relational Queries. *ACM Transactions on Database Systems*, 28(4):427–466, 2003.
- [CRS⁺07] Michael J. Cafarella, Christopher Ré, Dan Suciu, Oren Etzioni, and Michele Banko. Structured Querying of Web Text: A Technical Challenge. In *Proceedings of CIDR 2007*, pages 225–234, 2007.
- [DDF⁺90] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.
- [GUP06] José Galindo, Angélica Urrutia, and Mario Piattini. *Fuzzy databases: Modeling, Design, and Implementation*. Idea Group, 2006.
- [HLES06] Jonathon S. Hare, Paul H. Lewis, Peter G. B. Enser, and Christine J. Sandom. Mind the Gap: Another Look at the Problem of the Semantic Gap in Image Retrieval. In *Multimedia Content Analysis, Management and Retrieval 2006*, volume 6073 of *Proceedings of SPIE*. SPIE, 2006.
- [HPF⁺09] Ming Hua, Jian Pei, Ada W. C. Fu, Xuemin Lin, and Ho Fung Leung. Top-K Typicality Queries and Efficient Query Answering Methods on Large Databases. *The VLDB Journal*, 18(3):809–835, 2009.
- [IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A Survey of Top-K Query Processing Techniques in Relational Database Systems. *ACM Computing Surveys*, 40(4), 2008.
- [KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer*, 42(8):30–37, 2009.
- [Kie02] Werner Kießling. Foundations of Preferences in Database Systems. In *Proceedings of VLDB 2002*, pages 311–322. Morgan Kaufmann, 2002.
- [KT09] Ravi Kumar and Andrew Tomkins. A Characterization of Online Search Behavior. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 32(2):3–11, 2009.

- [LHC05] Bing Liu, Mingqing Hu, and Junsheng Chen. Opinion Observer: Analyzing and Comparing Opinions on the Web. In *Proceedings of WWW 2005*, pages 342–351. ACM, 2005.
- [LZLM07] Ying Liu, Dengsheng Zhang, Guojun Lua, and Wei-Ying Ma. A Survey of Content-Based Image Retrieval with High-Level Semantics. *Pattern Recognition*, 40(1):262–282, 2007.
- [Moe06] Marie-Francine Moens. *Information Extraction: Algorithms and Prospects in a Retrieval Context*. The Information Retrieval Series. Springer, 2006.
- [Mot88] Amihai Motro. VAGUE: A User Interface to Relational Databases that Permit Vague Queries. *ACM Transactions on Office Information Systems*, 6(3):187–214, 1988.
- [MS06] Imran R. Mansuri and Sunita Sarawagi. Integrating Unstructured Data into Relational Databases. In *Proceedings of ICDE 2006*. IEEE Computer Society, 2006.
- [Mur04] Gregory L. Murphy. *The Big Book of Concepts*. MIT Press, 2004.
- [RIK07] Tapani Raiko, Alexander Ilin, and Juha Karhunen. Principal Component Analysis for Large Scale Problems with Lots of Missing Values. In *Proceedings of ECML 2007*, volume 4701 of *LNAI*, pages 691–698. Springer, 2007.
- [RJ10] Brian Riordan and Michael N. Jones. Redundancy in Perceptual and Linguistic Experience: Comparing Feature-Based and Distributional Models of Semantic Representation. *Topics in Cognitive Science*, to appear, 2010.
- [Ros78] Eleanor Rosch. Principles of Categorization. In Eleanor Rosch and Barbara L. Lloyd, editors, *Cognition and Categorization*, pages 27–48. Lawrence Erlbaum, 1978.
- [SB10] Joachim Selke and Wolf-Tilo Balke. Extracting Features from Ratings: The Role of Factor Models. In *Proceedings of M-PREF 2010*, pages 61–66, 2010.
- [SS04] Alex J. Smola and Bernhard Schölkopf. A Tutorial on Support Vector Regression. *Statistics and Computing*, 14(3):199–222, 2004.
- [Sur04] James Surowiecki. *The Wisdom of Crowds*. Doubleday, 2004.
- [SWS⁺00] Arnold W. M. Smeulders, Marcel Worring, Simone Santini, Amarnath Gupta, and Ramesh Jain. Content-Based Image Retrieval at the End of the Early Years. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(12):1349–1380, 2000.
- [Wei07] Gerhard Weikum. DB & IR: Both Sides Now. In *Proceedings of SIGMOD 2007*, pages 25–29, 2007.
- [WKQ⁺08] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 Algorithms in Data Mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.
- [WKRS09] Gerhard Weikum, Gjergji Kasneci, Maya Ramanath, and Fabian Suchanek. Database and Information-Retrieval Methods for Knowledge Discovery. *Communications of the ACM*, 52(4):56–64, 2009.
- [ZG09] Xiaojin Zhu and Andrew B. Goldberg. *Introduction to Semi-Supervised Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool, 2009.
- [ZGBN07] Xuan Zhou, Julien Gaugaz, Wolf-Tilo Balke, and Wolfgang Nejdl. Query Relaxation using Malleable Schemas. In *Proceedings of SIGMOD 2007*, pages 545–556, 2007.

A Framework for Evaluation and Exploration of Clustering Algorithms in Subspaces of High Dimensional Databases

Emmanuel Müller ♦ Ira Assent ■ Stephan Günemann •
Patrick Gerwert • Matthias Hannen • Timm Jansen • Thomas Seidl •

♦ Karlsruhe Institute of Technology (KIT), Germany
emmanuel.mueller@kit.edu

■ Aarhus University, Denmark
ira@cs.au.dk

• RWTH Aachen University, Germany
{guennemann, gerwert, hannen, jansen, seidl}@cs.rwth-aachen.de

Abstract: In high dimensional databases, traditional full space clustering methods are known to fail due to the curse of dimensionality. Thus, in recent years, subspace clustering and projected clustering approaches were proposed for clustering in high dimensional spaces. As the area is rather young, few comparative studies on the advantages and disadvantages of the different algorithms exist. Part of the underlying problem is the lack of available open source implementations that could be used by researchers to understand, compare, and extend subspace and projected clustering algorithms. In this work, we discuss the requirements for open source evaluation software and propose the OpenSubspace framework that meets these requirements. OpenSubspace integrates state-of-the-art performance measures and visualization techniques to foster clustering research in high dimensional databases.

1 Introduction

In recent years, the importance of comparison studies and repeatability of experimental results is increasingly recognized in the databases and knowledge discovery communities. VLDB initiated a special track on *Experiments and Analyses* aiming at comprehensive and reproducible evaluations (e.g. [HCLM09, MGAS09, KTR10, SDQR10]). The conferences SIGMOD followed by SIGKDD have established guidelines for repeatability of scientific experiments in their proceedings. Authors are encouraged to provide implementations and data sets. While these are important contributions towards a reliable empirical research foundation, there is still a lack of open source implementations for many state-of-the-art approaches. In this paper we present such an open source tool for clustering in subspaces of high dimensional data.

Clustering is an unsupervised learning approach that groups data based on mutual similarity [HK01]. In high dimensional spaces, subspace clustering and projected clustering identify clusters in projections of the full dimensional space.

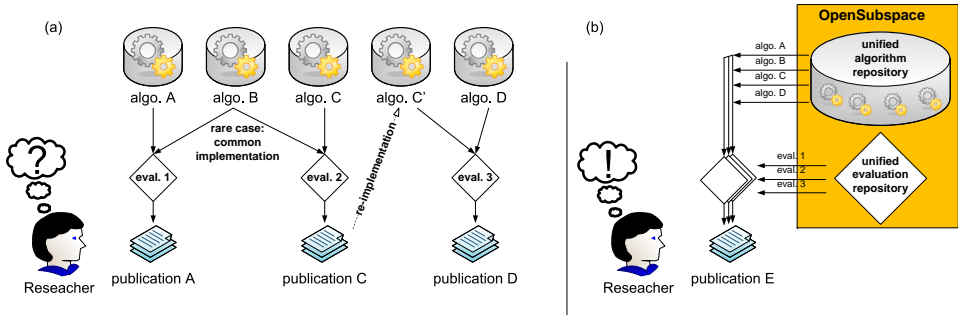


Figure 1: Subspace and projected clustering research with and without a general repository for data, algorithms, comparison, and evaluation.

It is a fundamental problem of unsupervised learning approaches that there is no generally accepted “ground truth”. As clustering searches for previously unknown cluster structures in the data, it is not known a priori which clusters should be identified. This means that experimental evaluation is faced with enormous challenges. While synthetically generated data is very helpful in providing an exact comparison measure, it might not reflect the characteristics of real world data. In recent publications, labeled data, usually used to evaluate the performance of classifiers, i.e. supervised learning algorithms, is used as a substitute [SZ04, KKRW05, AKMS07a]. While this provides the possibility of measuring the performance of clustering algorithms, the base assumption that clusters reflect the class structure is not necessarily valid.

Some approaches therefore resort to the help of domain experts in judging the quality of the result [KKK04, BKKK04, KKRW05]. Where domain experts are available, which is clearly not always the case, they provide very realistic insights into the usefulness of a clustering result. Still, this insight is necessarily subjective and not reproducible by other researchers. Moreover, there is not sufficient basis for comparison, as the clusters that have not been detected are unknown to the domain expert. This problem is even more aggravated in high dimensional subspace or projected clustering. As the number of results is typically huge, it is not easily possible to manually analyze the quality of different algorithms or even different runs of the same algorithm.

As there is no ground truth, nor accepted benchmark data or measures for evaluating subspace and projected clustering, the experimental evaluation can be hardly set into relation to other published results. Especially the results are incomparable, as there are no publicly available common implementations neither for subspace/projected clustering algorithms nor for evaluation measures (cf. Fig 1). As a consequence, progress in this research area is slow, and general understanding of the advantages and disadvantages of different algorithms is not established. The source code for experimental evaluation is most of the time implemented by the authors themselves and often not made available to the general public. This hinders further experimental study of recent advances in clustering. As tedious re-implementation is often avoided, only few comparisons between new proposals and existing techniques are published.

For clustering (but also for classification and association rule mining), the open source tool WEKA (Waikato Environment for Knowledge Analysis) has been very helpful in allowing

researchers to analyze the behavior of different algorithms in comparison [WF05]. It provides measures for comparison, visualization of the results, and lets researchers add their own algorithms and browse through the implementation of other techniques.

For subspace and projected clustering, such a general tool does not exist. In this paper, we discuss the requirements for a successful open source tool that supports evaluation and exploration of subspace and projected clustering algorithms and their cluster results. Our framework OpenSubspace fulfills these requirements by integration of measurement and visualization techniques for in-depth analysis. Furthermore, it will be useful in establishing benchmark results that foster research in the area through better understanding of advantages and disadvantages of different algorithms on different types of data. It includes successful techniques in demonstration systems for visualization and evaluation of subspace mining paradigms [MAK⁺08, AMK⁺08, GFKS10, GKFS10, MSG⁺10].

As anyone will be able to see the implementation, the code base can be continually revised and improved. Researchers may analyze the algorithms on a far greater range of parameter values than would be possible within the scope of a single conference or journal publication (cf. Fig 1). Based on this, we published a thorough evaluation study on subspace/projected clustering techniques [MGAS09]. As open source basis for this study, this publication provides an overview of techniques included in our OpenSubspace framework.

For scientific publications the open source implementations in OpenSubspace enable more fine grained discussions about competing algorithms on a common basis. For authors of novel methods OpenSubspace gives the opportunity to provide their source code and thus deeper insight into their work. This enhances the overall quality of publications as comparison is not based any more on incomparable evaluations of results provided in different publications but on a common algorithm repository with approved algorithm implementations. In Figure 1 we compare the current situation (on the left side) with the improved situation having a common repository of both subspace/projected clustering and evaluation measures (on the right side). Thus, OpenSubspace aims at defining a common basis for research and education purposes maintained and extended by the subspace/projected clustering community.

None of the existing data mining frameworks provide both subspace/projected clustering as well as the full analytical and comparative measures for the full knowledge discovery cycle. KNIME (Konstanz Information Miner) is a data mining tool that supports data flow construction for knowledge discovery [BCD⁺09]. It allows visual analysis and integration of WEKA. Orange is a scripting or GUI object based component system for data mining [DZLC04]. It provides data modeling and (statistical) analysis tools for different data mining techniques. Rattle (the R Analytical Tool To Learn Easily) is a data mining toolkit that supports statistical data mining based on the open source statistical language R [Wil08]. Evaluation via a number of measures is supported. In all of these frameworks subspace clustering or projected clustering are not included. ELKI (Environment for DeveLoping KDD-Applications Supported by Index Structures) is a general framework for data mining [AKZ08]. While it also includes subspace and projected clustering implementations, the focus is on index support and data management tasks. With respect to evaluation and exploration, it lacks evaluation measures and visualization techniques for an easy comparison of clustering results. Furthermore, as a stand alone toolkit it does not provide an integration into popular tools like WEKA.

2 Subspace and Projected Clustering

Clustering is an unsupervised data mining task for grouping of objects based on mutual similarity [HK01]. In high dimensional data, the “curse of dimensionality” hinders meaningful clustering [BGRS99]. Irrelevant attributes obscure the patterns in the data. Global dimensionality techniques such as Principle Components Analysis (PCA), reduce the number of attributes [Jol86]. However, the reduction may obtain only a single clustering in the reduced space. For locally varying attribute relevance, this means that some clusters will be missed that do not show in the reduced space. Moreover, dimensionality reduction techniques are unable to identify clusterings in different reduced spaces. Objects may be part of distinct clusters in different subspaces.

Recent years have seen increasing research in clustering in high dimensional spaces. Projected clustering aims at identifying the locally relevant reduction of attributes for each object. More specifically, each object is assigned to exactly one cluster (or noise) and a corresponding projection. Subspace clustering allows identifying several possible subspaces for any object. Thus, an object may be part of more than one cluster in different subspaces.

2.1 Paradigms

While subspace and projected clustering are rather young areas that have been researched for only one decade, several distinct paradigms can be observed in the literature. Our open source framework includes representatives of these paradigms to provide an overview over the techniques available. We provide implementations of the most recent approaches from different paradigms (cf. Fig. 2):

Subspace clustering

Subspace clustering was introduced in the *CLIQUE* approach which exploits monotonicity on the density of grid cells for pruning [AGGR98]. *SCHISM* [SZ04] extends *CLIQUE* using a variable threshold adapted to the dimensionality of the subspace as well as efficient heuristics for pruning. Both are grid-based approaches which discretize the data space for efficient detection of dense grid cells in a bottom-up fashion.

In contrast, density-based subspace clustering defines clusters as dense areas separated by sparsely populated areas. In *SUBCLU*, a density monotonicity property is used to prune subspaces in a bottom-up fashion [KKK04]. *PreDeCon* extends this paradigm by introducing the concept of subspace preference weights to determine axis parallel projections [BKKK04]. A further extension *FIREs* proposes an approximative solution for efficient density-based subspace clustering [KKRW05]. In *DUSC*, dimensionality bias is removed by normalizing the density with respect to the dimensionality of the subspace [AKMS07a]. Its extension *INSCY* focuses on efficient in-process removal of redundancy [AKMS08]. Recently, more general techniques have been proposed for optimization of the resulting set of clusters to eliminate redundant results and to include novel knowledge in orthogonal projections [MAG⁺09, GMFS09].

Projected clustering

Projected clustering approaches are partitioning methods that identify disjoint clusters in

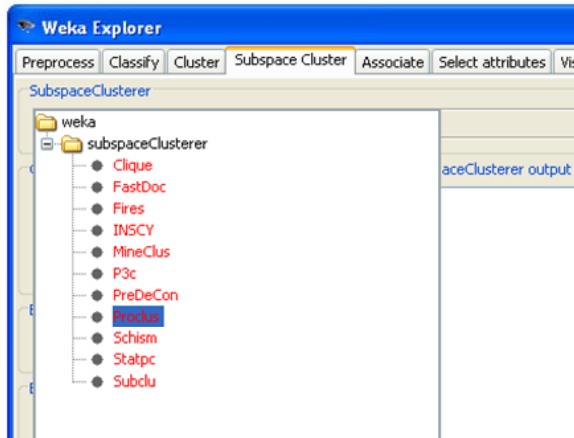


Figure 2: Algorithms implemented in OpenSubspace

subspace projections. *PROCLUS* extends the k-medoid algorithm by iteratively refining a full-space k-medoid clustering in a top-down manner [AWY⁺99]. *P3C* combines one-dimensional cluster cores to higher-dimensional clusters bottom-up [MSE06]. Its extension *StatPC* searches for non-redundant significant regions [MS08]. Further techniques are *DOC* a randomized approach using a Monte Carlo algorithm to find projected clusters represented by dense hypercubes [PJAM02] and *MineClus* an extension using the FP-tree for iterative projected clustering [YM03].

2.2 Challenges

Both subspace clustering and projected clustering pose new challenges to the mining task but especially to evaluation and exploration of the actual clustering results. In the following, we will show that these challenges have not yet been addressed by recent open source systems. Furthermore, they can not be solved by simply applying traditional techniques available for low dimensional clustering paradigms.

The WEKA framework provides several panels for different steps in the knowledge discovery cycle as well as for different data mining tasks (cf. Fig. 3). Besides structuring the GUI for users of the framework, the API reflects these different tasks in being structured according to classifiers, clustering algorithms, etc. This means that the Java class hierarchy reflects the common properties of each of the tasks. From this, several challenges arise in introducing a new data mining task, namely subspace/projected clustering, and new evaluation and visualization methods.

Due to special requirements in high dimensional mining we cannot simply extend the clustering panel in WEKA by adding new algorithms. We have to set up a new subspace panel by introducing techniques specialized to the new requirements in all areas (mining, evaluation and visualization). Subspace and projected clustering algorithms differ from clustering (or other data mining tasks such as classification) in that each cluster is associated with

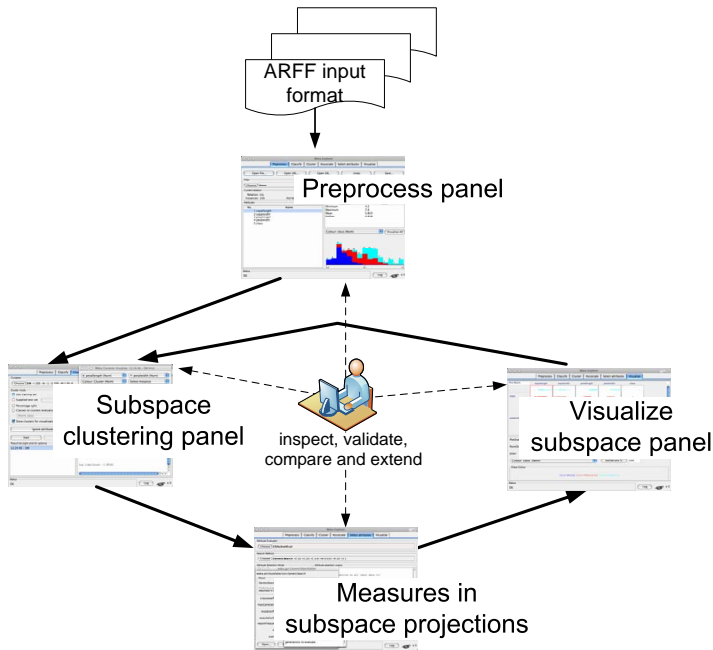


Figure 3: KDD cycle in WEKA for subspace and projected clustering

a possibly different subspace projection. As a consequence, the existing representation that assumes that all objects are clustered with respect to the initially chosen dimensions, is not valid. Moreover, the result is not necessarily partitioning. A single object may be part of several subspace clusters. These two aspects are important for the subspace/projected clustering panel, i.e. for the interface that describes common properties of these approaches. Moreover, these aspects have to be taken into consideration for evaluation and visualization as well. Following the same rationale, it is necessary to provide new APIs to allow meaningful analytical and visual tools in the OpenSubspace framework. Any measure that supports subspace and projected clustering evaluation needs to incorporate information on the respective subspace projection of each cluster. Visualization techniques to provide techniques for comparison of results in differing projections, as in [AKMS07b], cannot plug into existing visualization interfaces for traditional clustering in WEKA for the same reason.

As a consequence, OpenSubspace allows identifying any result with a corresponding set of dimensions, i.e. the subspace in which the result cluster resides. This is taken into consideration both for the subspace/projected clustering panel itself with the display of numeric results and evaluation measures, as well as for the visualization panel. This streamlined approach ensures that for all steps in the KDD cycle, representation in the correct subspace projection is achieved.

2.3 Lack of Ground Truth in Clustering

As briefly mentioned in the Introduction, clustering is a challenging task with respect to evaluation, as there is usually no ground truth. This means that it is in the very nature of clustering to search for yet unknown patterns in the data that provide novel and interesting insights to the user. Moreover, even for historic data, the true patterns that were interesting, are not known, either. This is in contrast to e.g. classification, where existing real world data can be used to easily validate the performance of any existing or newly presented classifier. Simply by checking the predicted class labels against the ones obtained from historic data, the classification accuracy can be easily measured in a reproducible fashion. In clustering, no such “labels” on historic data exists. Such “labels” would require an exhaustive enumeration of all combinatorial possibilities and their comparison. This is clearly infeasible even for reasonably small to medium datasets.

To allow for reproducible analysis, some publications resort to the measure of classification accuracy [BZ07, MAK⁺09]. The underlying idea is to find an objective measure for the performance of clustering. The assumption is that the class labels somehow reflect the natural grouping of the data, and can therefore be used to judge the performance of clustering algorithms as well. While this does provide some measure for comparison of these approaches, the underlying assumption is not necessarily valid and can even, in the worst case, produce random results. For example, an unsupervised clustering technique might detect a group of objects covering different labels, which might be meaningful as a clustering result. The given class labels, however, reflect only a single concept while clustering and especially subspace clustering aim at detecting multiple unknown concepts. The opposite case might happen as well: The clustering can split a set of objects with common labels into two clusters. Both meaningful clustering results are punished by evaluation measures simply based on the labels. As a consequence, class labels might provide only very limited insight into the performance of clustering algorithms.

Another approach taken in clustering evaluation is the use of synthetic data. Such artificial databases overcome the above mentioned problems by the generating process, the best clustering is already known. There are several limitations to this approach, however. First, most synthetic datasets are generated just for a single publication to evaluate the benefits of the proposed method. As such, they serve a very important purpose: they provide the means to understand whether the proposed method indeed detects (subspace or projected) clusters of the nature defined by the authors. Moreover, as the ideal clustering is known, the performance of algorithms perform on this dataset can be checked without having to resort to class labels. Even though some publications use very elaborate models to generate datasets that follow distributions that are believed to be observed in practical applications, there is obviously no guarantee that synthetic data is like real world data. Synthetic data, by its very nature, represents what is thought to occur in the datasets we analyze, but since we do not know which clusters might actually go unnoticed in real world data, these properties cannot be known.

Some researchers suggest using the help of domain experts in getting an informed answer to the quality of clustering results. Domain experts are obviously very helpful in judging the practical usefulness of the results and in ranking several possibilities in relation. However, as is true for the above examples, alternatives that are not known, i.e. not presented to the domain expert, cannot be taken into consideration. As a result, a very good cluster-

ing solution might be available and would be much more important to the domain expert. However, as this clustering is not retrieved, the domain expert cannot give a corresponding judgment. Moreover, manual analysis performed by domain experts is very limited to small result sizes and few parameter variations. Manual inspection of varied settings on a variety of datasets, as would be required for in-depth analysis, is clearly not feasible for humans. And, as mentioned before, the number of result clusterings in high dimensional spaces tends to grow enormously with the number of attributes. As a consequence, domain experts cannot judge typical outcomes of most subspace and projected clustering results. Moreover, the results indicated by domain experts are subjective and cannot be reproduced by other researchers.

As a consequence, any dataset used for evaluation is necessarily only a glimpse at the performance of (subspace or projected) clustering algorithms. As we will see later on, the open source idea provides a means to combine several of these glimpses into a larger picture towards an integrated view of clustering performance. As both the source code for validation of the results and the datasets are collected, a more integral picture is provided which can be easily extended by applying these algorithms to the datasets.

2.4 Lack of Standard Evaluation Measures in Clustering

Another problem in the evaluation of subspace and projected clustering lies in the evaluation measures themselves. This problem is closely related to the one of suitable datasets in that different results cannot be easily compared. Measuring the quality of (subspace and projected) clustering results is not straightforward. Even if the ground truth for any dataset were available, there are different ways of assessing deviations to this ground truth and of computing an overall performance score. For evaluation of clustering algorithms, large scale analysis is typically based on pre-labelled data, e.g. from classification applications [MSE06, BZ07]. The underlying assumption is that the clustering structure typically reflects the class label assignment. At least for relative comparisons of clustering algorithms, this provides measures of the quality of the clustering result.

In the literature, several approaches have been proposed. Quality can be determined as entropy and coverage. Corresponding roughly to the measures of precision and recall, entropy accounts for purity of the clustering (e.g. in [SZ04]), while coverage measures the size of the clustering, i.e. the percentage of objects in any subspace cluster. Open-Subspace provides both coverage and entropy (for readability, inverse entropy as a percentage) [AKMS07a]. Inverse entropy measures the homogeneity in the clustering result with respect to a class label. The measurement assumes a better clustering structure if the detected clusters are formed by objects homogeneously labeled with the same class labels. Besides the above mentioned problem of possible discrepancies in class labels and clustering structure, homogeneity of class labels is only one aspect of a good clustering structure. The coverage of the data set has to be measured separately to ensure that most of the objects occur in at least one cluster. Furthermore, overall homogeneity itself can be biased by many small homogeneous clusters dominating bigger inhomogeneous clusters.

Another approach is direct application of the classification accuracy. Accuracy of classifiers (e.g. C4.5 decision tree) built on the detected patterns compared with the accuracy of the same classifier on the original data is another quality measure [BZ07]. It indicates to

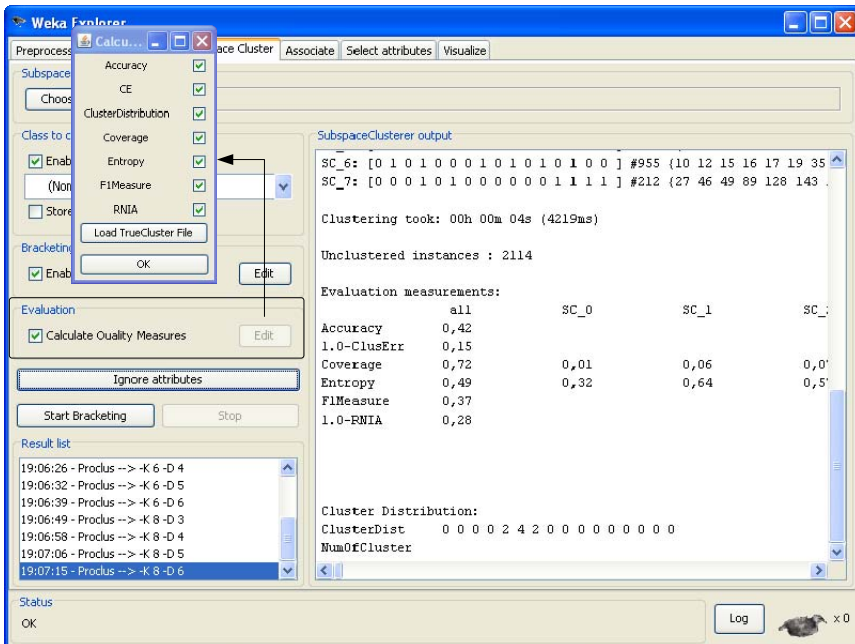


Figure 4: Evaluation measures in OpenSubspace

which extend the subspace clustering successfully generalizes the underlying data distribution. This approach is refined to enhanced classification accuracy which takes the original attributes and the ones that are derived as a combination of the original ones through the clustering. By comparing the performance of classifiers on the original attributes only with the performance of the same classifiers on original plus derived attributes, an insight into the quality of the clustering is achieved. However, as discussed before, any measured improvement is valid only with respect to the class labels. It is unclear, in which way the findings generalize to data without class labels.

The F1 value is commonly used in evaluation of classifiers and recently also for subspace or projected clustering as well [MSE06]. The F1 measure compares the clusters that are found by any particular (subspace or projected) clustering algorithm with an assumed ground truth by taking the harmonic mean of precision and recall. This approach obviously suffers from the same drawbacks as any class label-based method, yet, additionally, it is open to interpretation in high dimensional spaces. As clusters are detected in subspace projections, any deviation might be punished with respect to the subspace projection and with respect to the inclusion of positive and negative false alarms. However, the basis for comparison is not as straightforward, as it might seem. Depending on whether individual clusters or the entire clustering are used for the assessment, different results might be achieved. As a consequence, results based on variants of the F1 measure are not comparable across publications as one is using different F1 measure definitions.

All mentioned measures simply compare the detected groups of objects against the class label given for each object. Thus, these measures only provide a quality criterion for

object clusters as they ignore the detected subspaces in each cluster. More enhanced subspace clustering measures take also the detected subspaces into account and compare them against the possibly given relevant dimensions [PM06]. We have implemented such measures like Cluster Error (CE) and Relative Non-Intersecting Area (RNIA) in our framework. However, they require not only class labeled data for evaluation but also the relevant dimensions for each label. Such information is not provided in most real world data sets (e.g. used in classification task). Relevant dimensions are only available for synthetic data, as they are used and provided by the generators that hide subspace clusters in high dimensional spaces. Although both CE and RNIA achieve more detailed measurements as they take both objects and dimensions into account the missing ground truth is even more obvious for these measures.

In addition to this, several other measures have been used. In general, they all require some ground truth for assessing the performance of (subspace or projected) clustering algorithms. And, since several different subspace clusters might combine into a single “true” projected cluster, it is not always clear how to judge the result in its deviation from the postulated ground truth. Consequently, published results cannot be compared simply by their performance scores. Since there is no objective best measure for all approaches that is commonly agreed upon, researchers cannot compare different algorithms based solely on published results.

OpenSubspace provides the framework for using several, widely used, evaluation measures for subspace and projected clustering algorithms. In Figure 4 we present the evaluation output with various measures for comparing subspace clustering results. This allows easy extension of published results for various measures and direct comparison. Over time, as more and more results are available on different datasets and with respect to different evaluation measures, a benchmark background is built. It provides the means for in-depth understanding of algorithms and evaluation measures and fosters research in this area based on individual researcher’s findings.

3 OpenSubspace Framework

With OpenSubspace we provide an open source framework tackling the challenges mentioned in the previous section. By fully integrating OpenSubspace into the WEKA framework we build on an established data mining framework covering the whole KDD cycle: pre-processing, mining, evaluation and visualization of the results, additionally including user feedback to the mining algorithm to close the KDD cycle. With OpenSubspace we focus on the mining, evaluation and exploration steps in this cycle (cf. Fig. 5). Providing a common basis for subspace/projected clustering as a novel mining step we achieve a framework for fair comparison of different approaches. For evaluation and exploration of the subspace and projected clusters OpenSubspace provides various evaluation measures for objective comparison of different clustering results. Furthermore, OpenSubspace provides visualization methods for an interactive exploration. Please refer to our website where we document our ongoing work in this project. It also contains more detailed information about OpenSubspace, its usage and extension. In the following, we will give an overview on the major contributions of OpenSubspace to the subspace clustering community:

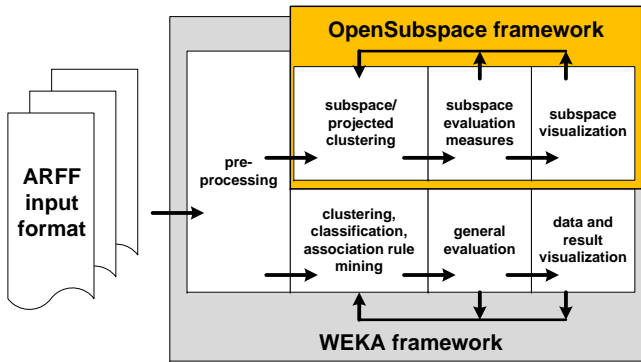


Figure 5: KDD cycle of OpenSubspace in WEKA

- Transparency of implementations
- Evaluation and comparison of algorithms
- Extensibility to further approaches

As we will show open source is the key property for all of these contributions. Open source code enables us to compare and validate the correctness of algorithm implementations. It gives us the basis for evaluating existing approaches on a common basis and leads thus to a fair comparison in future publications. By having the code of recent approaches at hand we enable the extension of existing algorithms to everyone and not only to the authors of these approaches.

3.1 Transparency of Implementations

The basis for thorough and fair evaluation is a common basis for all implementations. OpenSubspace provides such a basis for data access supporting both main and secondary storage. Furthermore, the framework provides a common interface for subspace clustering implementations. Algorithms which extend this interface can be easily plugged into OpenSubspace.

All of these algorithms are provided as open source. This transparency of the underlying implementation ensures high quality algorithms in the framework. The research community is able to review these implementations according to the original publication of the algorithm. Even improved versions can be provided which go beyond the descriptions in the publications using novel data structures, heuristics or approximation for specialized purposes. The benefit of reviewing implementations based on open source is especially useful as in most publications authors can only sketch their algorithms. This makes it difficult to re-implement such approaches. Various different interpretations of one approach could arise if only closed implementations were available. Using these different implementations of the same approach leads to incomparable results in scientific publications as evaluations have different bases. Open source repositories as in OpenSubspace prevent

such differing implementations that might even be biased and does away with the need for re-implementation for competing approaches. Overall OpenSubspace aims at a transparent and thus fair basis for evaluations of various approaches for detecting meaningful subspace clusters.

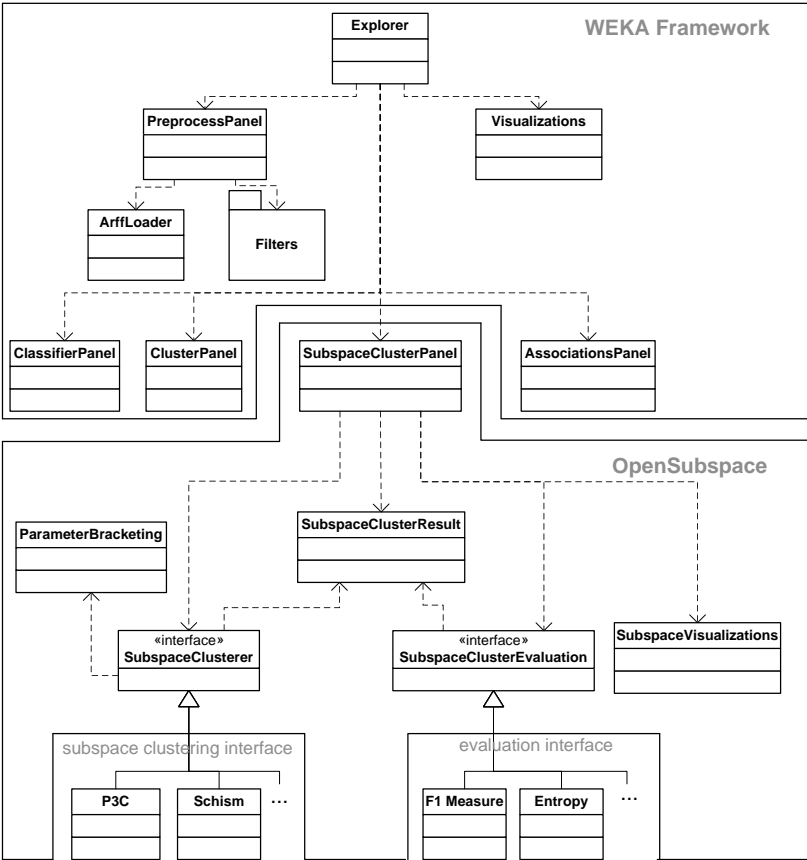


Figure 6: UML class diagram of the OpenSubspace framework

For extending the OpenSubspace algorithm repository our framework incorporates two open interfaces, which enable extensibility to further subspace clustering algorithms and new evaluation measurements. In Figure 6 we show the main classes of our OpenSubspace framework which extends the WEKA framework by a new subspace clustering panel.

Subspace clustering shows major differences compared to traditional clustering; e.g. an object can be part of several subspace clusters in different projections. We therefore do not extend the clustering panel, but provide a separate subspace clustering panel.

Recent subspace clustering algorithms described in Section 2.1 are implemented based on this framework. The abstraction of subspace clustering properties in OpenSubspace allows to easily add new algorithms through our new subspace clustering interface.

3.2 Evaluation and Comparison of Algorithms

Given the framework with transparent implementations of subspace clustering algorithms OpenSubspace enables researchers to evaluate their methods against competing approaches available in our repository. We establish a basis for developing new methods to perform an objective evaluation on arbitrary subspace clustering algorithms. OpenSubspace defines evaluation measurements based on labeled data sets. It includes measurements like entropy, coverage, F1-value, Cluster Error, RNIA and accuracy used in recent subspace/projected clustering publications as a basis for thorough evaluations. The set of measures is formally defined in our recent evaluation study providing additional experiments and analyses on several clustering paradigms [MGAS09].

In OpenSubspace all of these evaluation methods are implemented and published as open source as well. For a fair and comparative evaluation these measurements have to be accessible to all researchers. Review and refinement of these measurements is essential as there is always the possibility of different interpretations of these measures. As a ground truth is not given for subspace clustering the data mining community has to develop new evaluation measures that rate the quality of different approaches. This seems to be as difficult as the mining task itself. Therefore, we do not only provide several evaluation techniques (cf. Section 2.4) to measure the quality of the subspace clustering, but also an open interface (cf. Fig. 6) to implement new measures. Further measures can be added by our evaluation interface, which allows to define new quality criteria for subspace clustering on a common basis for all algorithms.

Evaluation measures summarize the result set in typically one real valued rating; however, visualization of results for more insight might be interesting. OpenSubspace, therefore, includes specialized visualizations for subspace clustering results with the possibility for interactive exploration. As stated before, subspace/projected clustering algorithms typically provide overwhelming result sets. Investigating these results is sometimes as difficult as looking at the raw data. For some specialized or domain dependent mining tasks it is even more important to investigate the actual clustering than to compare it with competing approaches. OpenSubspace provides specialized visualization techniques which close the KDD cycle by providing user feedback (cf. Fig. 5). Our framework provides interactive exploration of the results and thus the opportunity to refine the mining step by exploring different parameter settings and their resulting clustering output [MAK⁺08, AMK⁺08]. In addition the different views detected by subspace clustering approaches can be visualized and explored as well [GFKS10, GKFS10].

3.3 Visualization Techniques

OpenSubspace provides visualization techniques to present subspace clustering results such that users can easily gain an overview of the detected patterns, as well as an in-depth understanding of individual subspace clusters and their mutual relationship.

Gaining a meaningful overview is crucial in allowing users to assess the overall subspace clustering result. As mentioned, subspace clustering is inherently challenging as both the typical number of resulting subspace clusters is usually enormous as well as that clusters

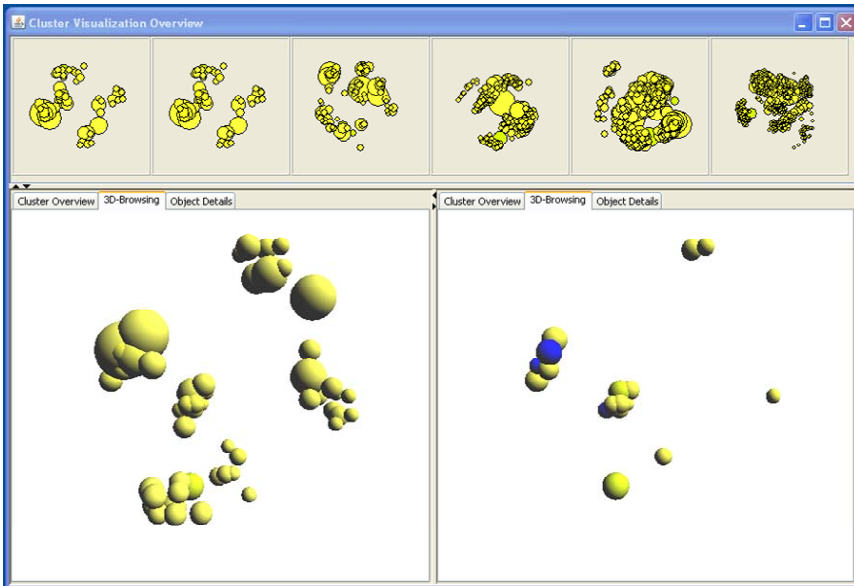


Figure 7: Visualization in OpenSubspace

in different projections are difficult to understand. Visualization techniques that were developed for full space clustering results rely on a common representation, i.e. no subspace projections [AKMS07b]. Consequently, they cannot be applied to subspace clustering.

Our framework thus contains specialized techniques for visualization of subspace clustering. 2d and 3d models are an adequate representation for human cognitive abilities. Based on a recently developed comparison measure for subspace clusters our system provides an overview on the entire subspace clustering result by MDS (multidimensional scaling) plots in both 2d and 3d [AKMS07b]. As illustrated in Figures 7, MDS approximates distances in high dimensional spaces by two or three dimensions. While the 2d representation is a static view that allows for easy reading, the 3d MDS plots allow users to interactively explore the overall subspace clustering result. They may move around the 3d representation to focus on those subspace clusters they are most interested in. At any point, they may choose individual subspace clusters in the plot to obtain more detailed information.

Thus, our MDS plots provide an overview on subspace clustering. Moreover, it helps users in interactively determining the best parameter setting. For any subspace clustering algorithm, some core parameters tend to have a large influence on the resulting output. We therefore present a bracketing representation, i.e. a series of 2d MDS plots for different parameters. Users thus get a clear visual impression of the effect of parameters and may choose the best ones for a feedback loop that generates the desired subspace clustering.

For in-depth analysis of any subspace clustering algorithm, representation of the key features of subspace clusters in a cognitively meaningful way is crucial. As subspace clustering results represent patterns in different projections by their very nature, visualization should contain information on the respective subspaces, the cluster values and additional

information on the interestingness measures computed by the subspace clustering algorithm. We use a color-coding scheme where the different axis in the HSV color space are used to represent different aspects of subspace clusters in a very compact and easy to understand manner [AKMS07b]. For easy navigation, subspace clusters can be zoomed into, and understood using a color legend on values of the subspace clusters.

3.4 Interactive Exploration

The conceptual design for interactive exploration in OpenSubspace is based on the *Visual Exploration Paradigm* [Kei02]: Starting from an overview over the subspace clustering result the user can navigate through the visualized patterns. By selecting subjectively interesting subspace clusters, the user may then obtain more detailed information where desired. Detailed information is provided on three levels: for entire subspace clusterings, for single subspace clusters, as well as for individual objects. Based on the discovered knowledge, the user can give feedback to the system for further improved results. This feedback loop enables the system to use the cognitive abilities of humans for better parameter settings and thus for meaningful subspace clusters.

Overview Browsing. Interactive exploration starts from an overview of all mined subspace clusters in which the user can browse. The automatically detected patterns are thus presented to the user for a general impression of the result and a comparison of the resulting clusters. As clusters are detected in arbitrary subspaces, they cannot be compared based on the full space dimensionality. We thus incorporate a distance function that takes the main characteristics of subspace clusters, their subspace dissimilarity and object dissimilarity into account for visualization in an MDS plot [AKMS07b]. Based on such an overall distance function, subspace clusters can be intuitively represented as circles in a 2d or 3d space (Figure 7). This approximation of the original high dimensional information to a 2d or 3d representation, allows human users to easily understand the result just by the visual impression. We enriches this MDS information by additional visual features. The size of a subspace cluster is represented as the diameter of the circle. Its dimensionality is encoded by the color of the circle. This information allows users to identify similar subspace clusters, those clusters of similar dimensionality, or of similar size, or to study the overall distribution of these characteristics in the result for further analysis.

Parameter Bracketing. Parameter setting is in general a difficult task for most data mining algorithms. This is especially the case for unsupervised techniques like clustering, where typically no prior knowledge about the data is available. This inherent problem of clustering is even more present in subspace clustering as the user has to provide parametrization for detecting clusters in different subspaces. In general the problem can be solved by guessing a parameter setting, looking at the result and then trying to choose a better parameter setting. To speed up this tedious process for users and give them more information to base their parameter choice on, we compute and visualize a series of different subspace clustering results at once, called bracketing of different parameter settings for direct feedback. This means that users obtain a series of MDS plots (cf. upper part of Figure 7) from which they pick the most appropriate one(s) for subsequent runs of the subspace clustering algorithms. By directly comparing the results of different parameter settings, parametrization is no longer a guess, but becomes an informed decision based

on the visual analysis of the effects of parameters. Moreover, this process is far more comfortable for users and allows reaching the desired subspace clustering result in fewer steps.

Direct subspace cluster comparison. For a more detailed analysis of two different parameter settings the user can select two clusterings out of the presented series of MDS plots by clicking on them in the bracketing representation. These two subspace clusterings are then presented as larger plots in the lower part of the cluster overview screen. Once again, detailed information for the subspace clusters can be obtained by picking individual subspace clusters.

3d Browsing. For the overview browsing we provide static 2-dimensional MDS plots. These static views provide a fixed perspective for easy comparison. For in-depth browsing, where focusing to different parts of the subspace clustering is of interest, a flexible navigation through MDS plots is provided. 3-dimensional MDS plot browsing allows users to shift, rotate and zoom into the MDS plot using standard 2-dimensional input devices or 3-dimensional input devices that allow for even more intuitive navigation. The user may thus identify similar or dissimilar subspace clusters that are of specific interest. In Figure 7, we show two 3-dimensional MDS plots representing two clustering results.

Interactive Concept Detection. In general, subspace clustering techniques were developed for the task of finding clusters in differing subspaces. Even more challenging is the grouping of clusters according to their specific concepts, for example the clusters 'smokers', 'joggers', or 'vegans' are manifestations of the concept 'health awareness'. Some recent approaches focus already on the task of grouping objects according to underlying concept structures [CFD07, GMFS09, GFMS10]: they find clusters in strongly differing subspace projections, providing the key for discovering the inherent concept structure. However, since the concepts are generative, i.e. they actually induce the clusters, they cannot be automatically concluded out of clusters. Accordingly, the mentioned subspace clustering techniques achieve concept-based aggregations of objects but are not capable of abstracting from these aggregations in the sense of named concepts.

In real-world applications, however, the interest lies in the explicit discovery and naming of the underlying concepts. This task cannot be solved automatically by unsupervised learning methods as subspace clustering but requires the domain knowledge of an expert. OpenSubspace supports the user in revealing the concepts out of a given subspace clustering [GFKS10, GKFS10]. It therefore provides the user with concept-oriented cluster visualization and interactive exploration to enable him to uncover the inherent concept structures. Each concept can be described by its occurring clusters on the one hand and its characteristic attributes on the other hand. Since the related clusters are not known beforehand, the idea is to capture the concepts through the structure of relevant attributes of the clustering. The relevant attributes are of particular importance for a semantic labeling of clusters and concepts. In the OpenSubspace framework, the user can take a closer look at the concept compositions and one can give feedback to refine or to recalculate the concept structures. Thus, the whole process of concept discovery in OpenSubspace is iterative and highly dependent on user interaction.

3.5 Extensibility of OpenSubspace

As a novel framework OpenSubspace provides the basis for further research. There are several algorithms implemented in our subspace/projected clustering repository. For evaluation measures we have included recently used measures in this field [SZ04, MSE06, BZ07, MAK⁺09, AKMS08]. However, as subspace clustering has just started to become a broader research topic, these evaluation measures can be only seen as first steps that are likely to be extended greatly in the near future. We included different visualization techniques in OpenSubspace which we presented in recent demonstration systems [MAK⁺08, AMK⁺08, GFKS10, GKFS10, MSG⁺10].

All three areas (mining, evaluation and visualization with interactive exploration) can be extended by open interfaces. Due to the fact that the whole framework is given as open source code it is easy to develop new algorithms, evaluation measures and visualizations. For researchers who wish to develop their own novel algorithm in this field we provide an easy way to integrate their approach into our framework and to perform a fair evaluation with competing approaches. Thus it is a key property of OpenSubspace to define an open basis for the development of new approaches, evaluation and visualization techniques.

We used and still use our framework for subspace clustering research but also for education in advanced data mining courses. In both cases we got positive feedback from our students who enjoyed easy and wide access and the predefined interfaces in our framework. Furthermore, we got encouraging feedback also by the community for our recent demonstration system which integrates extensible mining techniques into WEKA.

4 Conclusion

With OpenSubspace we provide an open source framework for the very active research area of subspace clustering and projected clustering. The aim of our framework is to establish a basis for comparable experiments and thorough evaluations in the area of clustering on high dimensional data. OpenSubspace is designed as the basis for comparative studies on the advantages and disadvantages of different subspace/projected clustering algorithms.

Providing OpenSubspace as open source, our framework can be used by researchers and educators to understand, compare, and extend subspace and projected clustering algorithms. The integrated state-of-the-art performance measures and visualization techniques are first steps for a thorough evaluation of algorithms in this field of data mining.

5 Ongoing and Future Work

OpenSubspace can be seen as the natural basis for our next task. We plan to develop evaluation measures that meet the requirements for a global quality rating of subspace clustering results. Evaluations with the given measurements show that none of the measurements can provide an overall rating of quality. Some measurements give contradicting quality ratings on some data sets. Such effects show us that further research should be done in this area.

Visualization techniques give an overall impression on the groupings detected by the algorithms. However, further research of meaningful and intuitive visualization is clearly necessary for subspace clustering. The open source framework for subspace mining algorithms has already encouraged researches in Visual Analytics and Human Computer Interaction to work on more meaningful visualization and exploration techniques.

For an overall evaluation framework OpenSubspace provides algorithm and evaluation implementations. However, further work has to be done to collect a bigger test set of high dimensional data sets. On such a benchmarking set one could collect best parameter settings for various algorithms, best quality results and screenshots of subspace clustering result visualizations as example clusters on these data sets. The aim of an overall evaluation framework with benchmarking data will then lead to a more mature subspace/projected clustering research field in which one can easily judge the quality of novel algorithms by comparing it with approved results of competing approaches.

Acknowledgment

We would like to thank the authors of SUBCLU, FIRES and MineClus for providing us with the original implementations of their approaches, which we adapted to our framework.

References

- [AGGR98] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *SIGMOD Record*, 27(2):94–105, 1998.
- [AKMS07a] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. DUSC: Dimensionality Unbiased Subspace Clustering. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 409–414, 2007.
- [AKMS07b] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. VISA: visual subspace clustering analysis. *SIGKDD Explorations*, 9(2):5–12, 2007.
- [AKMS08] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. INSCY: Indexing Subspace Clusters with In-Process-Removal of Redundancy. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 719–724, 2008.
- [AKZ08] Elke Achtert, Hans-Peter Kriegel, and Arthur Zimek. ELKI: A Software System for Evaluation of Subspace Clustering Algorithms. In *Proc. International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 580–585, 2008.
- [AMK⁺08] Ira Assent, Emmanuel Müller, Ralph Krieger, Timm Jansen, and Thomas Seidl. Pleiades: Subspace Clustering and Evaluation. In *Proc. European conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pages 666–671, 2008.
- [AWY⁺99] Charu C. Aggarwal, Joel L. Wolf, Philip S. Yu, Cecilia Procopiuc, and Jong Soo Park. Fast algorithms for projected clustering. *SIGMOD Record*, 28(2):61–72, 1999.

- [BCD⁺09] Michael R. Berthold, Nicolas Cebon, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. KNIME - the Konstanz information miner: version 2.0 and beyond. *SIGKDD Explorations*, 11(1):26–31, 2009.
- [BGRS99] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When Is “Nearest Neighbor” Meaningful? In *Proc. International Conference on Database Theory (ICDT)*, pages 217–235, 1999.
- [BKKK04] Christian Böhm, Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density Connected Clustering with Local Subspace Preferences. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 27–34, 2004.
- [BZ07] Björn Bringmann and Albrecht Zimmermann. The Chosen Few: On Identifying Valuable Patterns. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 63–72, 2007.
- [CFD07] Ying Cui, Xiaoli Z. Fern, and Jennifer G. Dy. Non-redundant Multi-view Clustering via Orthogonalization. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 133–142, 2007.
- [DZLC04] Janez Demar, Bla Zupan, Gregor Leban, and Tomaz Curk. Orange: From Experimental Machine Learning to Interactive Data Mining. In *Proc. Knowledge Discovery in Databases (PKDD)*, pages 537–539, 2004.
- [GFKS10] Stephan Günemann, Ines Färber, Hardy Kremer, and Thomas Seidl. CoDA: Interactive Cluster Based Concept Discovery. In *Proc. of the VLDB Endowment*, pages 1633–1636, 2010.
- [GFMS10] Stephan Günemann, Ines Färber, Emmanuel Müller, and Thomas Seidl. ASCLU: Alternative Subspace Clustering. In *Proc. MultiClust Workshop at ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010.
- [GKFS10] Stephan Günemann, Hardy Kremer, Ines Färber, and Thomas Seidl. MCEXplorer: Interactive Exploration of Multiple (Subspace) Clustering Solutions. In *Proc. IEEE International Conference on Data Mining (ICDM)*, 2010.
- [GMFS09] Stephan Günemann, Emmanuel Müller, Ines Färber, and Thomas Seidl. Detection of orthogonal concepts in subspaces of high dimensional data. In *Proc. ACM Conference on Information and Knowledge Management (CIKM)*, pages 1317–1326, 2009.
- [HCLM09] O. Hassanzadeh, F. Chiang, H.C. Lee, and R.J. Miller. Framework for evaluating clustering algorithms in duplicate detection. *Proc. of the VLDB Endowment*, 2(1):1282–1293, 2009.
- [HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.
- [Jol86] Ian Jolliffe. *Principal Component Analysis*. Springer, New York, 1986.
- [Kei02] Daniel A. Keim. Information Visualization and Visual Data Mining. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):1–8, 2002.
- [KKK04] Peer Kröger, Hans-Peter Kriegel, and Karin Kailing. Density-Connected Subspace Clustering for High-Dimensional Data. In *Proc. SIAM International Conference on Data Mining (SDM)*, pages 246–257, 2004.
- [KKRW05] Hans-Peter Kriegel, Peer Kröger, Matthias Renz, and Sebastian Wurst. A Generic Framework for Efficient Subspace Clustering of High-Dimensional Data. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 250–257, 2005.

- [KTR10] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proc. of the VLDB Endowment*, 3(1):484–493, 2010.
- [MAG⁺09] Emmanuel Müller, Ira Assent, Stephan Günnemann, Ralph Krieger, and Thomas Seidl. Relevant Subspace Clustering: Mining the Most Interesting Non-Redundant Concepts in High Dimensional Data. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 377–386, 2009.
- [MAK⁺08] Emmanuel Müller, Ira Assent, Ralph Krieger, Timm Jansen, and Thomas Seidl. Morpheus: interactive exploration of subspace clustering. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1089–1092, 2008.
- [MAK⁺09] Emmanuel Müller, Ira Assent, Ralph Krieger, Stephan Günnemann, and Thomas Seidl. DensEst: Density Estimation for Data Mining in High Dimensional Spaces. In *Proc. SIAM International Conference on Data Mining (SDM)*, pages 173–184, 2009.
- [MGAS09] Emmanuel Müller, Stephan Günnemann, Ira Assent, and Thomas Seidl. Evaluating clustering in subspace projections of high dimensional data. *Proc. of the VLDB Endowment*, 2(1):1270–1281, 2009.
- [MS08] Gabriela Moise and Jörg Sander. Finding non-redundant, statistically significant regions in high dimensional data: a novel approach to projected and subspace clustering. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 533–541, 2008.
- [MSE06] Gabriela Moise, Jörg Sander, and Martin Ester. P3C: A Robust Projected Clustering Algorithm. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 414–425, 2006.
- [MSG⁺10] Emmanuel Müller, Matthias Schiffer, Patrick Gerwert, Matthias Hannen, Timm Jansen, and Thomas Seidl. SOREX: Subspace Outlier Ranking Exploration Toolkit. In *Proc. European conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pages 607–610, 2010.
- [PJAM02] Cecilia M. Procopiuc, Michael Jones, Pankaj K. Agarwal, and T. M. Murali. A Monte Carlo algorithm for fast projective clustering. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 418–427, 2002.
- [PM06] Anne Patrikainen and Marina Meila. Comparing Subspace Clusterings. *IEEE Transactions on Knowledge and Data Engineering*, 18(7):902–916, 2006.
- [SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. of the VLDB Endowment*, 3(1):460–471, 2010.
- [SZ04] Karlton Sequeira and Mohammed Javeed Zaki. SCHISM: A New Approach for Interesting Subspace Mining. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 186–193, 2004.
- [WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, USA, 2005.
- [Wil08] Graham Williams. Rattle: A graphical user interface for data mining in R using GTK. R package version 2.3.115. <http://rattle.togaware.com/>, 2008.
- [YM03] Man Lung Yiu and Nikos Mamoulis. Frequent-pattern based iterative projected clustering. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 689–692, 2003.

Efficient Interest Group Discovery in Social Networks using an Integrated Structure/Quality Index [◇]

Adriana Budura ^{*} Sebastian Michel [‡] Karl Aberer ^{*}

^{*} Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland
adriana.budura@epfl.ch, karl.aberer@epfl.ch

[‡] Universität des Saarlandes, Saarbrücken, Germany
smichel@mmci.uni-saarland.de

Abstract: We consider the problems of interest group discovery in a social network graph using term-based topic descriptions. For a given query consisting of a set of terms, we efficiently compute a connected subset of users that jointly cover the query terms, based on the annotation vocabulary utilized by users in the past. The presented approach is twofold; first we identify so-called seed users, centers of interest groups, that act as starting points of the group exploration. Subsequently, we inspect the seed users' neighborhoods and build up the tree connecting the most promising neighbors. We demonstrate the applicability and efficiency of our method by conducting a series of experiments on data extracted from a Web portal showing that our method does not only provide accurate answers but calculates these also in an efficient way.

1 Introduction

In this work, we consider a graph of users with edges reflecting a certain degree of closeness, for instance in terms of explicit friendship links in social networking sites such as Flickr or Facebook, or derived from relations, such as the *is-co-author* relation in the case of bibliographic information. A common feature of these scenarios is that users exhibit a certain amount of profile information, either explicitly by specifying keyword based areas of interest, or implicitly by using specific terms when annotating resources or by assigning keywords to publications, as is common practice. Given this information, we aim at providing means to efficiently identify connected subgroups of users whose profiles match a particular keyword-based query.

Many approaches have modeled user behavior in online social networks, with the observation that users usually annotate resources they are interested in and that such annotations represent a summary of a user's interests [SSMY08]. Complementary to the information centric functionality of these portals, most of them provide means to create social communities in form of groups and friendship links. Analogously, in a bibliographic information system, links between authors can be defined based on the co-author relation, and semantic annotations to authors are given by keywords they have used in their publications. A query would involve certain keywords, such as *databases*, *graph*, and *algorithm*, with the goal to identify a set of authors, connected by the co-author relation, which jointly cover the

[◇] This work is partially supported by NCCR-MICS (grant number 5005-67322), the FP7 EU Project OKKAM (contract no.ICT-215032), and the German Research Foundation (DFG) Cluster of Excellence "Multimodal Computing and Interaction" (MMCI).

three query terms. Clearly, the assigned keywords reflect the authors' (research) interests, hence, we denote these groups as *interest groups*.

Identifying users which have used at least one of the query terms in the past is straight forward as it requires a simple per-term indexing mechanism in form of sets or more advanced inverted files where users are ordered by some kind of quality information. This would allow us to readily apply well established methods to return single users that are relevant w.r.t. our query. However, as it is often unlikely that single users can answer the query as a whole, the general answer to a query is a subset of users that together cover the whole query. As the number of such user groups can potentially be very large, we have to introduce a scoring function which assesses for a given group its suitability to the query, hence, being able to return the top-k list of user groups. Such a scoring function consists of two ingredients: (i) the quality of the users contained in the group and (ii) the compactness of the group. This is related to the problem of keyword search on graphs, or graphs imposed by the primary/foreign key relation ship in relational databases (cf., e.g., [HP02, HWY07, BHN⁺02]) which has been extensively studied in the past years.

In this work we will use a combined approach that integrates both user quality information and structural (network) information to make the computation tractable. For sake of readability we will use the terms user, friendship, and tags, in this paper.

1.1 Model and Problem Statement

We consider a directed friendship graph $G = (V, E)$ with nodes representing the users and with an edge $e = (u_i, u_j) \in E$ if user u_i is a friend of user u_j .

Each user is furthermore associated with a set of documents she has annotated so far, hence, we have for each user a set of $(tag, frequency)$ -pairs that reflect her tagging history. The frequency number is simply a count of how many times the user has used a certain tag.

We assume users to issue tag-based queries to explore the social network. For each of these queries the task is to return the most relevant and compact groups whose users jointly cover the specified query.

Our coverage requirement can be formalized as follows: Given a query as a set of tags $Q = \{t_1, t_2, \dots, t_n\}$ and a set of users $U = \{u_1, u_2, \dots, u_m\}$, we consider that U covers Q iff $Q \subseteq \bigcup_{u \in U} T_u$, where T_u is the set of tags used by user u . This notion of coverage can be relaxed, by accepting groups that only partially cover the query. This is in particular useful if the query is very large and does not have any results, which can be a consequence of restricting the compactness of the returned groups. We nevertheless favor those subsets of users that answer a larger fraction of the query.

Concerning the quality of the answer we need to address two requirements: (i) *relevance*: the returned interest group should be relevant w.r.t. the query terms and (ii) *compactness*: the returned interest group should be as small as possible (in terms of number of users) and the users in this group should be connected as closely as possible.

1.2 Contribution and Outline

The contribution of this paper can be summarized as follows: (i) We present an approach to detect communities of interest in a social network; (ii) We integrate the friendship information into the content information for efficient query processing; (iii) We evaluate our approach using a real world dataset, taken from the popular delicious portal.

Section 2 gives an overview of existing related work. Section 3 contains the scoring model for communities and a first glance on the problem of identifying user groups that cover a query. Section 4 explains how to get a handle on the community selection by integrating network information into standard index lists. Section 5 shows how we employ a threshold algorithm over the extended index lists to identify centers of communities. Section 6 presents our graph traversal algorithm that computes community trees based on previously selected seed users. Section 7 contains the performance evaluation and Section 8 concludes the paper.

2 Related Work

As collaborative tagging sites become more and more popular, many approaches to exploit this data for information retrieval have been proposed. Li et al. [LGZ08] propose a method which uses tags to discover clusters of users that share common social interests. In a first phase, association rules mining algorithms are used to discover patterns of tag occurrence which identify higher level topics. In a second step, users and URLs are clustered according to these topics.

Another research area close to our work is that of community discovery in networks which can be summarized as the problem of dividing a graph into sub-groups of nodes such that the nodes in a partition are densely connected among each other and less connected with the rest of the network [NG04, FLGC02, YL08]. In general, all these approaches rely solely on the topological information (i.e., links between nodes) in the network in order to discover densely connected communities, whereas in this work we take a query-driven view on this problem which is guided by the content of the nodes. We do not require that our users form a densely connected sub-graph per-se, as we allow for communities to span over several topic clusters, depending on the nature of our query.

Other approaches such as [CZC08, CCL⁺09] only rely on textual information about users in a social networking site to identify communities. As opposed to the above mentioned approaches we also integrate the topological information in the network.

Recently, Li et al. [LNL⁺08] introduce an algorithm for community discovery in large text collections which builds a hierarchy of communities based on the relationship between textual documents (i.e., links within these documents) as well as on the content of the documents. In the first phase, community cores are identified based on the topological information solely and in the second phase the communities are identified based on the textual information by studying the latent topic distributions in the documents. As opposed to this method, our algorithm is driven by the user query and we integrate the quality of the users and the link structure in the same computation.

Qin et al. [QYCT09] proposes algorithms to enumerate all or a top-k set of communities using foreign key relations in a RDBMS. Similar to our work, they introduce an upper bound for the community (i.e., group) size and deal with extracting centers of communities. The focus is put on a higher level algorithmic solution, disregarding performance issues on the level of index accesses to determine the top centers.

Lappas et al. [LLT09] consider the problem of finding a group of experts in social networks that together cover a certain set of given skills. Their approach is limited to a boolean assignment of skills to users, leaving no room for a fine grained differentiation, which is in particular essential for our tag based detection mechanism. The authors sketch to use thresholds of skill values to obtain a binary assignment. While this seems to be reasonable in scenarios with limited sets of skills, such an approach would introduce, in our setup, one threshold per tag, therefore making a manual tuning impossible. Instead,

we treat quality information about tag usages as a first class citizen, used not only in the scoring function, but also in the way we select promising centers of interest groups.

Recently, Sozio and Gionis [SG10] presented an approach to find subgraphs given a set of nodes as a query that have to be contained in the answer graph, i.e., aim at identifying a community around a given set of users. Limiting the search space with this input is in contrast to our approach which aims at extracting communities based on semantic descriptions of the users by inspecting the global graph.

For the index creation, we make use of existing work from the area of keyword search over graphs (cf., e.g., [HP02, HWY07, BHN⁺02]) which has been extensively studied in the past years. In particular, the way we create our index is similar to the concept of keyword-node lists in [HWY07] where for any keyword a list is created containing for each node the distance to the keyword, i.e., the distance of a user to the user that has used the tag in the past, in our scenario. We extend these lists to also contain quality information on frequency of tag usages, and maintain for each node and for a configurable number of distances, one list (per-tag, per-distance) representing for each user the best quality score to be found in its neighborhood with the given distance. This enables the application of a two level threshold algorithm over these lists.

3 Scoring Model

Before delving into the details of our approach we will introduce below the scoring model which we use in order to select the most promising user groups that cover a specified query. As already sketched above, there are two ingredients that we require to be reflected in the scoring model: (i) the relevance of the group w.r.t. the query terms, and (ii) the compactness of the group.

In order to assess the relevance of the group we consider the tagging behavior of the users that are part of that group. More precisely, we employ a standard mechanism by relying on the tag frequencies (tf) of the users for the tags that belong to the query. Given a set of users $U = \{u_1, u_2, \dots, u_m\}$ that cover a query Q the score of a user u_i w.r.t. the query can be expressed as the sum of tag frequencies for every term of the query that also belongs to the tag set of the given user: $s(u_i, Q) = \sum_{t \in Q \wedge t \in T_{u_i}} tf(u_i, t)$.

The compactness of a group is assessed in terms of the number of edges in the smallest tree that connects all users in the specified community. To combine both relevance scores and compactness scores we make use of a weighted sum. Given a group of users U which covers the query Q and given a spanning tree that connects the users in U , $G(E, U')$ with edges E and users U' where $U \subset U'$, the score of this group is computed as

$$s(U, Q) := \alpha * \sum_{u_i \in U} s(u_i, Q) + (1 - \alpha) * \frac{1}{|E| + 1} \quad (1)$$

The first part of the scoring formula represents the quality of the users in the group w.r.t. the query, while the second part reflects its compactness. In this work we opted for using the edge count of the spanning tree as an indicator for compactness, but other choices, like taking the radius of the tree is a potential measure, too. The weighting parameter α is used to give more weight to the tag score than to the compactness, or vice versa. The main reason for introducing this parameter is that the decision whether a returned interest group is good or bad is highly subjective; one could prefer to settle for larger trees if the connected users are particularly promising in terms of tag scores. On the other hand, one could prefer

trees that are as compact as possible, even at the expense of a lower tag-based quality score.

A naive way to use the scoring model now to find the best communities is to execute a brute-force enumeration of all possible sets of users that jointly cover the query tags, considering all users that are associated with at least one tag out of the query tag set. For each set of users we would then generate the minimal tree connecting all users in that set and assess its utility w.r.t. our scoring model. This method would indeed calculate the “best” tree that covers the query tags, however, it is prohibitively expensive due to: (i) the extremely large number of candidate user sets to be assessed, and (ii) the compactness assessment function which involves calculating the so-called Steiner tree for each given user set.

We could choose to address these two subproblems separately – first find the most relevant users w.r.t. the query and then return the smallest subgraph which connects all of them. However, it is obvious that when adopting this solution we will end up with users being far away from each other, albeit being of high quality w.r.t. the tags. These kind of results do clearly not correspond to our desired solution.

In our approach, we deal with both requirements at the same time – construct the algorithm in such a way as to return compact trees that contain the “best” possible nodes, according to our scoring function.

4 Index Creation

Assume for each tag a list of $(user, tf)$ -pairs sorted by tf (i.e., the number of (distinct) documents the user has annotated with this tag). This resembles the basic inverted index paradigm from standard Information Retrieval that can be efficiently used to compute the most relevant users w.r.t. a query by applying Fagin’s TA algorithm or variants like the NRA algorithm [FLN03]. The number of index list entries is actually much smaller than in traditional document retrieval tasks, as we only deal with entries that represent the tag sets of each user and this is at least one order of magnitude lower than the number of documents, even for large networks. Instead of using the plain tag frequency count (tf), we can plug in any tf based variant, such as $tf \cdot idf$, as the choice is independent of our algorithmic solution.

In order to combine the compactness and the relevance information, we will integrate the network structure into the traditional inverted index lists by propagating the tags along the edges of the user graph. As a result, we will express the pairwise distances among users solely in terms of inverted lists of users and tags, which allow us to compute the relevance of a user w.r.t. a query, by implicitly considering the user’s neighborhood.

Below we give a more detailed description on how the tag propagation is implemented. Subsequently, Section 5 will focus on the query processing task, which uses Fagin’s threshold algorithm to identify what we call *seed users*, promising centers of relevant groups, solely working on tag specific lists of $(user, tf)$ -pairs, which have been enhanced through our tag propagation mechanism.

4.1 Propagating Tagging Behavior

For each $(user, tag)$ -pair we compute the minimum distance from the given tag to the user, based on the user graph. If a user holds a tag, then the distance between this user and the tag is 0. However, if a user does not hold a tag, but has a neighbor that holds that tag the distance between that user and the tag will be computed w.r.t. the distance between the neighbor and the initial user, similar to the concept of keyword-node lists in [HWY07]. However, not only the distances but also the quality (frequency of tag usages) is taken into

account. In this way, each user will inherit the tagging behavior of her neighbors and will act as an indicator of the suitability of her neighborhood w.r.t. the tags in the query.

We will propagate the tags in this manner over λ hops in the friendship graph. From the implementation point of view, our so-called λ -extensions consist of additional index lists, more precisely λ additional index lists per query term. For each $(user, tag)$ -pair and for each distance ($\leq \lambda$) we store the maximum inherited tf score from any neighbor that can be found within that distance. Obviously, the lists for $\lambda = 0$ correspond to the original index lists.

We propagate tags only through inlinks, i.e., links from so-called “idols” to “fans”, and not in the opposite direction, in order to maintain the implicit trust relationship defined by the users through friendship links (i.e., if a “fan” chose another user to be his “idol” we assume that she is prepared to follow the tagging behavior of the latter).

More formally, for a user u and her set of tags T_u we compute the score for any tag t as follows

$$score(u, t) = \max_{u' \in U \wedge |path(u, u')| \leq \lambda} (tf(u', t))$$

We precompute for each tag an index list of *all* users and their corresponding score w.r.t. that tag (including the tags and tf values inherited from their neighbors, up to a distance λ).

5 Query Processing

For the query evaluation over the λ -enriched index lists we employ Fagin’s NRA algorithm, one of the standard threshold algorithms, which uses only sorted (sequential) accesses to the lists. The result of this computation will be a set of top- k *seed* users which represent the “centers” of our interest groups. Since these users are chosen according to the λ -enriched lists, we are sure that in their close neighborhood we will find a set of users that covers our query. Furthermore, we implicitly assess the quality of the user tree and estimate its size, directly through the seed user retrieval process.

Note that the number of tags per query is assumed to be quite large, hence, we opted for a disjunctive query evaluation mode where we do not require all tags to be present at the top- k most suitable users. This avoids empty results as it can happen that we cannot identify a user group that covers the query as we restrict the diameter of the returned group to $2 * \lambda$. However, we still favor users which cover a larger number of tags, although their tag frequency values might not be very large.

To use this algorithm, we normalize the tag based scores by $1 / \sum_s$ where \sum_s is the sum over all original tf scores. In addition, for each non-zero score observed in a list, we add a value of 1 to favor users that have many query tags over those with few but high-score tags. In particular, this means, that given an aggregated score s for a user we know that she contributed to $\lfloor s \rfloor$ tags. Contributing to a tag does not mean that she actually owns the tags (i.e., has annotated documents herself), it means that $\lfloor s \rfloor$ tags can be found in her λ -neighborhood.

Due to the λ -extension, we keep for each tag t and value of λ a separate index list $L_{t, \lambda}$ which contains $(userid, tf)$ -pairs for tag t sorted by tf in descending order. The tf values in these lists are basic scores inherited from the λ -neighborhood, with no score adjustments applied so far as this is defined at runtime using Equation 1 (i.e., the database contains only the raw information). This scoring function is applied on the fly, which causes no problems as it is order preserving and computationally trivial.

With a given score $s_{raw} := L_{t,\lambda}(u)$ and λ we adjust this score according to the following formula:

$$score(s_{raw}, \lambda) := \alpha * s_{raw} + (1 - \alpha) * \frac{1}{\lambda + 1} \quad (2)$$

In the initialization step, for each tag $t \in Q$ and for each value of λ a sequential read to the database is opened. In total there are $|Q| * \lambda$ index lists to be accessed. The algorithm performs a round robin read over the index list groups (grouped by tag) and inside each tag group reads an entry from the list with the currently highest score, as in [TSW05].

Hence, overall, for a user u and a query Q the final score is given by:

$$score(u, Q) := \sum_{t \in Q} \max_{\lambda} \{score(L_{t,\lambda}(u))\} \quad (3)$$

where $score(L_{t,\lambda}(u))$ denotes the score of user u in the index list for tag t for given λ . This scoring function follows the goal of finding seed users, i.e., users whose neighborhood contains a tree with “optimal” score w.r.t. the query (cf., Equation 1).

Following the standard principles of the NRA algorithm as explained in the beginning of this section, during the sequential scans we maintain for each observed user a score range given by lower and upper bound scores, denoted as W and B , respectively.

To calculate B for a particular user w.r.t. a tag t (note that B is computed for those tags for which we don't know the user's tf values yet), we look at the current scan line scores for the lists corresponding to t $L_{t,\lambda=0} =: \tau_0, \dots, L_{t,\lambda=l-1} =: \tau_{l-1}$, where l denotes the number of lists per tag. Note that these scores are obviously known and that the algorithm always reads from the list that provides the largest score. We then calculate the best possible score by looking at the τ_i values, considering their λ values and applying the aggregation function, i.e., $\hat{\tau} := \max_{i \in \{0, \dots, l-1\}} \{score(\tau_i, i)\}$.

Let the set $E(u)$ denote all tags for which a user u has been observed in *any* of the index lists, $\hat{\tau}(t)$ the max possible score for a tag t , and W the *worstscore* as explained above, then the *bestscore* B for that user can be calculated as:

$$B(u, Q) := W(u, Q) + \sum_{t \in Q - E(u)} \hat{\tau}(t) \quad (4)$$

While the aggregation function mentioned in Equation 2 is monotone it does not correspond to Equation 1 which is easy to see. However, it gives the correct bound w.r.t. Equation 2 that defines the computed scores in the index lists.

6 Spanning Tree Computation

Given a set of query tags Q and a user u that has been selected as a *seed* user as explained in the previous section our next task is to return the interest group around this user that covers our query and assess its compactness. We treat each *seed* user independently and therefore consider only a single user in this section.

Assume $\tau = score(u, Q)$ to be the score of user u w.r.t. Q given the aggregation model above. It is clear that in the λ neighborhood of the user we will observe $\lfloor \tau \rfloor$ distinct tags

$t_1, \dots, t_{\lfloor \tau \rfloor}, t_i \in Q$. Our task is to identify the most relevant users holding these tags and we solve this through an iterative process.

We start by executing the query on the λ neighborhood N of the *seed* user and select the highest ranked user u according to the scoring model. Then the query is reduced by those tags answered by user u and the user is added to a set of terminal users, that contains those users that will be returned by our method. This process is iterated until the set of remaining query tags is empty, or has size smaller than $|Q| - \lfloor \tau \rfloor$. For the terminal users the pairwise distances are retrieved from the database (we recompute these up to level $2 * \lambda$) and the minimum spanning tree (MST) is computed, which is known to be a 2-approximation of the true Steiner tree.

Note that when computing the top- k *seed* users we do not have any information attached to the λ index list entries that indicates the exact size of our final trees. We only know that a seed user represents a tree with a set of edges $|E| \leq \sum_i \lambda_i$. The implication of this is that lower ranked seed users can indeed provide better trees, as the upper bound of the tree size might not be tight enough. We will see this behavior in the experimental evaluation when we observe that higher ranked seed users are not necessarily superior to lower ranked seed users. However, the variation in terms of tree size and tree tag score is relatively small.

7 Experimental Evaluation

We have implemented our algorithm in Java 1.6 and executed on a Windows 2003 server with a quad core 2.33 GHz Intel Xeon CPU, 16GB RAM, and a 800GB RAID-5 disk. The data is stored in an Oracle 11g database in form of (userid, tag, score)-entries, separated in different tables for different values of λ with B+ indexes on (tag, score DESC, user) for fast access. The user graph is kept in main memory. The precomputed pairwise user distances up to distance 2 are kept in the database, too, with an B+ index on (user1, user2, distance) for random reads.

We use a partial crawl of the *delicious*¹ portal which consists of approximately 120,000 html pages, annotated by 13,515 users with 59,143 distinct tags. As our approach is purely annotation based, we disregard the content of the annotated pages and solely focus on (i) the friendship graph among users and (ii) the tagging behavior of users. For (ii), we consider for each user and each tag t the frequency with which the user has annotated pages with t . As there are often users which use particular tags very rarely, we introduce a threshold value to disregard these very rarely used tags. We will see its interpretation later.

In order to run our experiments we need to propagate the tags along the links of the friendship graph, as presented in Section 4. We propagate the tagging behavior only within a certain number of hops λ , where we pick a value of $\lambda = 2$ for this paper, which is sufficient, given the small world property and that this value actually means that we are considering user groups with a diameter of 4.

In order to generate the queries for testing our approach, we have parsed the high level categories for three different topics out of *Open Directory Project* (*dmoz.org*), for instance *dmoz.org/Health/* for the topic *health*, ending up with the following topics with corresponding numbers of terms (in parenthesis): *Computers* (72), *Health* (61), *Physics* (35). We assemble queries by randomly selecting an equal number of terms from each of these topics. The total number of terms is varied in the experiments.

We run our experiments on a set of 100 queries and vary the following parameters: (i) α - the parameter of our scoring function (c.f., Equation 1), (ii) *min tf* - a threshold used

¹www.delicious.com

to study the influence of a threshold based (i.e., tag count) noise filter on the result quality and fraction of queries that can be answered, (iii) the number of terms in the queries.

We report on the following measures: (i) *Interest group (tree) score*: this measure reflects the combined quality and compactness score of the returned user group and is computed according to Equation 1. (ii) *Number of edges*: this is the size of the returned user group in terms of number of edges. (iii) *Query response time*: to get a better insight for the runtime of our algorithm, we measure the query response time and split it into the time needed to identify the seed users and the time needed to build the trees in the subsequent step.

7.1 Results

In our experiments we start by generating the top- k seed users for a given query and then return the best user tree around each seed. When we report on the rank, we always refer to the rank of the seed that returned a given user tree and not to the rank of the tree itself. In order to make the results comparable we only take into consideration those queries that are fully answered by all ranks of our seed nodes (except for the experiments corresponding to Figure 3 (right)). Except for the cases where we vary these parameters explicitly, we fix the value of α to 0.5 (in order to put the same weight on both terms in the scoring function), the number of terms in the query to 21 (in order to better underline the performance of our method) and the value of the *min tf* threshold to 1. We also apply a standard smoothing on the tag score part as due to standard normalization issues it is not comparable to the compactness score (in terms of different magnitudes of the score values).

Figure 1 (left) shows the average number of edges for different ranks and *min tf* thresholds. Recall that this threshold specifies the tag frequency value from which on a user is considered for a particular tag. The increasing trend in the figure follows the intuition that with a higher value for this threshold, the number of suitable users decreases, which, in turn, increases the expected number of edges in the answer tree. Clearly, the biggest impact of the threshold is when going from 1 to 5, changing the average tree size (i.e., the size of the returned user group) from 3.7 to 7.7. The changes for higher values of the threshold are less dramatic, but show the same trend. The fact that lower ranked seed users lead to better trees is already explained in Section 6.

Figure 1 (right) reports on the overall tree score for different ranks of *seed* users, when varying the *min tf* threshold. The scores of the trees generated by the top-5 seed users are very close, which indicates the fact that we deal with many equally suitable interest groups around these seeds. We also observe a clear increasing score trend with a bigger *min tf* value. This is expected since a bigger tag frequency for our users should lead to a better overall quality of the trees. On the other hand, as we observed from Figure 1 (left), we would expect the number of edges in the trees to increase with *min tf* and this should bring the score lower. However, we can observe that if the *min tf* values improve from 5 to 15 this means an improvement by a factor of three in the sum of *tf* values (the first term of our scoring function); from Figure 1 (left) we can see that for such an improvement we only have to pay a penalty of adding on average one extra edge. In other words, the overall score of the trees increases with *min tf* because the penalty of adding extra edges is entirely compensated by a large improvement factor of the *tf* scores.

Figure 2 (right) shows the quality values of the users w.r.t. different α values. This corresponds to the first term of our scoring function and contains the sum of tag frequency values for the users contained in the group. We can observe an increasing trend which means that with a bigger α we give less weight to the tree size factor and, therefore, ex-

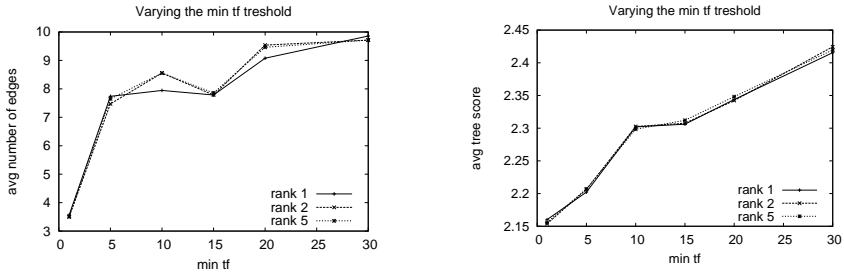


Figure 1: Varying $\min tf$, showing the number of edges (left) and the score (right) for different ranks ($\alpha = 0.5$, nr of terms/query = 21).

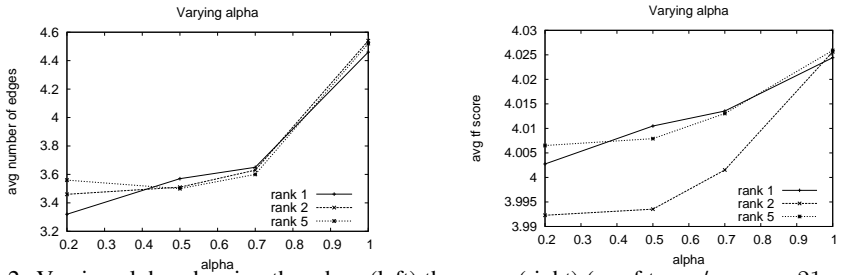


Figure 2: Varying α , showing the edges (left) the score (right) (nr of terms/query = 21, $\min tf = 1$).

plore bigger interest groups in order to increase the quality of the potential nodes. When looking at different ranks we can see that the first rank tends to dominate the others.

In Figure 2 (left) we see the impact of α on the average number of edges for different ranks. The variation of α has a clear impact, as the trend shows for all ranks a larger tree size for larger values of α . For $\alpha = 1.0$ the trees are the largest, which is intuitive as this means that the tree size does not matter at all in the score calculation, hence, users with a high tag score component are selected, even though they are quite far away from each other. This is also reflected in Figure 3 (left) which reports on the average number of edges with varying rank, plotted for different values of α .

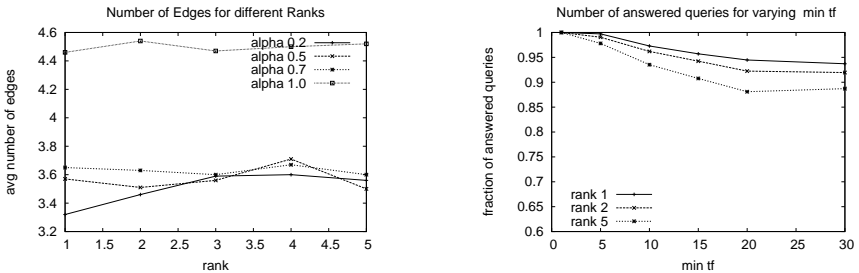


Figure 3: Left: Varying the rank, showing the edges (nr of terms/query = 21, $\min tf = 1$). Right: Fraction of answered queries for increasing $\min tf$ threshold ($\alpha = 0.5$, nr of terms/query = 21).

Figure 3 (right) shows the fraction of completely answered queries for different values of the *min tf* threshold. Since we allow only users with a certain number of occurrences for a particular tag to be selected, at a certain point, a query cannot be answered if the users are less than λ edges away, which explains the decreasing trend. Nevertheless, the fraction of full query answers is still big (88% for a *tf* threshold of 30), even for large threshold value.

7.2 Performance Study

We have measured the average query response time when varying the number of query tags, i.e., the query size for a number of 5 queries and for $\alpha = 0.5$ and *min tf* = 1 (the plot has been omitted due to space constraints). As expected, the total query response time increases with increasing query size. To get a better understanding where the time is spent, we look at two ingredients separately, the time to identify the seed users and the time needed to build the user tree in the seed users' λ -neighborhood. As we can see, the time to identify the seed users is clearly the dominating factor and also increases with increasing query size. The time to build the tree is almost negligible and, furthermore, remains almost constant with varying query size, varying from around 800ms for 6 tags to 1200ms for 27 tags.

7.3 Baseline Comparison

We also conducted an experiment to compare our approach to a baseline using a small subset of the original dataset (1000 users and the friendship links between them) as it is not possible to apply the baseline method to a reasonably large graph. The baseline method generates all possible subsets that cover a given query and ranks them based on our scoring model. In order to compute the compactness scores we use the same 2-approximation of the Steiner tree as for our method, which is based on pair-wise distances between nodes. We ran our algorithm on the same subset of nodes and we compare our best achieved results with the results of the baseline. Figure 4 reports on query response time, tree score, and number of edges for a set of 50 queries, when varying the number of query tags. Considering the query response time, Figure 4 (left) shows a linear scale-up of our approach with the query size whereas the cost of the baseline grows exponentially. This is not a surprise as the baseline exhaustively inspects all possible combinations of users that cover the query tags. Looking at the tree score in Figure 4 (right), we see that we are a constant factor away from the true answer and that the performance of our algorithm does not degrade with larger queries. The same happens for the number of edges (plot omitted due to space constraints).

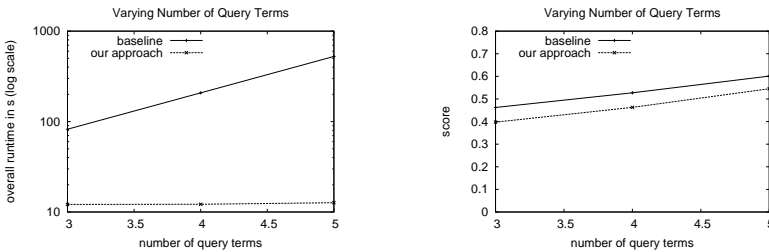


Figure 4: Query response time (left) and tree score (right) for changing number of tags in the queries.

8 Conclusion

We have presented an approach to identify interest groups in a social network based on the tagging behavior of users. Our approach adapts a computationally expensive graph problem to the common framework of top- k threshold algorithms following existing work on keyword search on graphs, for an efficient query execution in order to find the “best” groups. The rationale behind this approach is to propagate tagging behavior along edges of the social friendship network, i.e., users inherit tagging behavior from their neighbors, to transform the community identification task into the problem of selecting single users. We have conducted an experimental analysis of the proposed algorithm using data obtained from a partial crawl of *delicious.com* to demonstrate the suitability of the presented framework.

References

- [BHN⁺02] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. *ICDE*, 2002.
- [CCL⁺09] WenYen Chen, Jon-Chyuan Chu, Junyi Luan, Hongjie Bai, Yi Wang, and Edward Y. Chang. Collaborative filtering for orkut communities: discovery of user latent behavior. *WWW*, 2009.
- [CZC08] WenYen Chen, Dong Zhang, and Edward Y. Chang. Combinational collaborative filtering for personalized community recommendation. *KDD*, 2008.
- [FLGC02] Gary William Flake, Steve Lawrence, C. Lee Giles, and Frans Coetzee. Self-Organization and Identification of Web Communities. *IEEE Computer*, 35(3), 2002.
- [FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4), 2003.
- [HP02] Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. *VLDB*, 2002.
- [HWY07] Hao He, Haixun Wang, Jun Yang 0001, and Philip S. Yu. BLINKS: ranked keyword searches on graphs. *SIGMOD*, 2007.
- [LGZ08] Xin Li, Lei Guo, and Yihong Eric Zhao. Tag-based social interest discovery. *WWW*, 2008.
- [LLT09] Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. *KDD*, 2009.
- [LNL⁺08] Huajing Li, Zaiqing Nie, Wang-Chien Lee, C. Lee Giles, and Ji-Rong Wen. Scalable community discovery on textual data with relations. *CIKM*, 2008.
- [NG04] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, Feb 2004.
- [QYCT09] Lu Qin, Jeffrey Xu Yu, Lijun Chang, and Yufei Tao. Querying Communities in Relational Databases. *ICDE*, 2009.
- [SG10] Mauro Sozio and Aristides Gionis. The community-search problem and how to plan a successful cocktail party. *KDD*, 2010.
- [SSMY08] Julia Stoyanovich, Amer-Yahia Sihem, Cameron Marlow, and Cong Yu. Leveraging Tagging Behavior to Model Users’ Interests in *del.icio.us*. *AAAI Spring Symposium on Social Information Processing*, Stanford University, 2008.
- [TSW05] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. Efficient and self-tuning incremental query expansion for top- k query processing. *SIGIR*, 2005.
- [YL08] Bo Yang and Jiming Liu. Discovering global network communities based on local centralities. *TWEB*, 2(1), 2008.

Filtertechniken für geschützte biometrische Datenbanken

Christian Böhm^a, Ines Färber^b, Sergej Fries^b, Ulrike Korte^c, Johannes Merkle^d,
Annahita Oswald^a, Thomas Seidl^b, Bianca Wackersreuther^a, Peter Wackersreuther^a

^a LMU München, {boehm|oswald|wackersb|wackersr}@dbs.ifi.lmu.de

^b RWTH Aachen, {faerber|fries|seidl}@informatik.rwth-aachen.de

^c BSI Bonn, ulrike.korte@bsi.bund.de

^d secunet Essen, johannes.merkle@secunet.com

Abstract: In immer mehr sicherheitsrelevanten Bereichen werden biometrische Erkennungstechniken für die Zugangskontrolle oder die Identitätsfeststellung einer Person eingesetzt. Da biometrische Merkmale hoch sensibel sind, müssen sie vor unbefugtem Zugriff geschützt werden. Sogenannte Template Protection Verfahren ermöglichen eine biometrische Authentisierung, ohne dass sich diese Merkmale aus den gespeicherten Referenzdaten ermitteln lassen. Allerdings erschweren diese Verfahren die Suche nach passenden Referenzdaten und machen daher die Identifikation innerhalb umfangreicher Datenbestände ineffizient. In diesem Artikel werden erste Ansätze untersucht um auch für große Datenmengen eine Identifikation auf Basis von geschützten Fingerabdrücken durchführen zu können. Die vorgestellten Verfahren erstellen durch Filtertechniken und Indexstrukturen eine geeignete Priorisierung der Datenbankeinträge, sodass der aufwändige exakte Vergleich zwischen Anfrage und den transformierten Einträgen gezielt erfolgen kann.

1 Einleitung

Der Einsatz biometrischer Merkmale in Identifikationssystemen hat in den letzten Jahren stark zugenommen. Biometrische Erkennungsmerkmale sind in der Regel universell, einzigartig, persistent und personengebunden. Die Persistenz der biometrischen Daten bedingt jedoch, dass sie einmal korumpiert unwiederbringlich als Identifikationsmerkmal für das betreffende Individuum verloren sind. Zudem bergen biometrische Merkmale neben den benötigten Informationen für eine Identifikation auch sehr sensible Informationen, z.B. über die ethnische Zugehörigkeit oder den Gesundheitszustand. Daher ist die Verwendung biometrischer Daten aus Sicht des Datenschutzes nicht unumstritten.

Um die sensiblen Daten sicher zu speichern, haben sich sogenannte *Template Protection Verfahren* etabliert, wobei das Fuzzy Vault-Verfahren [JS06] dabei zu den Bekanntesten zählt. Hier werden die biometrischen Eigenschaften durch künstlich hinzugefügte Merkmale gegen weit verbreitete Angriffstechniken geschützt. Da eine Suche in den transformierten Referenzdaten im Allgemeinen dann aber sehr ineffizient ist, ist der praktische Einsatz für Identifikationszwecke bisher noch ein offenes Problem.

In der Publikation von Korte *et al.* [KMN09] wurde bereits ein Verfahren zum Abgleich eines ungeschützten Fingerabdrucks mit einem geschützten Datenbankeintrag basierend

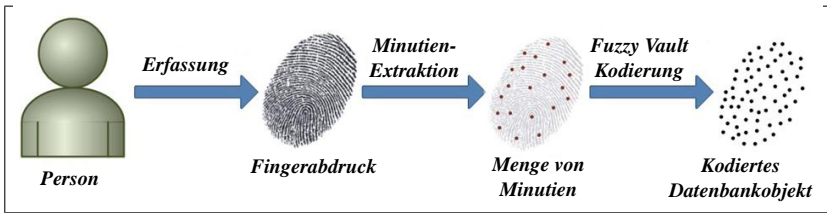


Abbildung 1: Erzeugen eines kodierten Datenbankobjekts für den Identifikationsprozess.

auf Minutien vorgestellt. Dieses System implementiert jedoch nur den Authentifikationsprozess, bei dem die Identität des Nutzers *a priori* bekannt ist. Um auch die Identifikation in akzeptabler Zeit zu beantworten, ist neben einer effizienten Verifikation auch die Anzahl der in Frage kommenden Datenbankobjekte geeignet einzuschränken. Indexstrukturen sowie Filterarchitekturen ermöglichen eine entsprechende Vorauswahl durch verschiedene Approximationstechniken. Allerdings unterliegen die biometrischen Daten neben der schützenden Transformation meist starkem Rauschen. So können die Finger beim Scannen gedreht oder verschoben aufgelegt werden. Zudem kann der Abdruck durch unterschiedlichen Druck des Fingers auf den Sensor Verzerrungen aufweisen. Zusätzlich können Deletionen oder Insertionen von Merkmalen auftreten. Es muss also davon ausgegangen werden, dass prinzipiell kein Objekt mit Sicherheit für die anschließende Verifikation ausgeschlossen werden kann.

Bisher sind keine effizienten Suchverfahren für eine Identifikationslösung mit Template Protection Verfahren bekannt. Diese Arbeit beschreibt daher erste Ansätze, um effiziente Datenbanktechniken in das biometrische Anwendungsgebiet zu integrieren. Durch den Einsatz effizienter Filterarchitekturen bzw. Indexstrukturen ermöglichen wir eine schnelle Identifikation von Personen auf Basis geschützter Fingerabdrucksbilder (vgl. Abbildung 1), wobei die Herausforderung darin liegt, mit einem sehr starken Rauschen innerhalb der Daten, verursacht durch das Fuzzy Vault und den zusätzlichen Rauscheffekten, umzugehen. Zwei Fingerabdrücke werden dabei auf Basis ihrer zuvor extrahierten Minutien (End- und Verzweigungspunkte der Papillarlinien) verglichen. Wir stellen zwei unterschiedliche Ansätze vor, die eine Rangfolge der Datenbankobjekte erstellen, sodass tendenziell ähnlichere Objekte für den anschließenden Verifikationsvorgang priorisiert werden. Unsere Experimente zeigen, dass sich die Zahl der durchzuführenden Verifikationen deutlich reduzieren lässt.

Der Rest dieses Artikels ist wie folgt gegliedert: Abschnitt 2 erläutert den allgemeinen Identifikationsvorgang. Unsere beiden Verfahren GeoMatch und BioSimJoin werden in Abschnitt 3 vorgestellt. Abschnitt 4 beschreibt den Prozess der Erstellung aller verwendeten biometrischen Datenbanken. Eine ausführliche Evaluierung unserer Verfahren folgt in Abschnitt 5. Abschließend fassen wir diesen Artikel in Abschnitt 6 zusammen.

2 Identifikationssysteme für Biometrische Datenbanken

Bei der biometrischen Identifikation steht die Identität des Benutzers nicht à priori fest. Sie entspricht daher einem Scan über die komplette Datenbank, bis eine Übereinstimmung gefunden wird. Als biometrische Merkmale eines Fingerabdrucks verwenden wir die kartesischen Koordinaten seiner Minutien $m = (m_x, m_y)$. Im Folgenden wird ein Finger stets durch die Menge seiner Minutien repräsentiert. Sei die Anfrage repräsentiert durch ein Template Q , das gegen eine Datenbank $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$ von Referenztemplates verglichen werden soll. Alle Referenztemplates R_j werden schon beim Enrolment durch das Fuzzy-Vault Verfahren transformiert. Im Folgenden wird zunächst das Enrolment, also das Erstellen von R_j , sowie die anschließende Verifikation von Q , beschrieben.

Enrolment. Das Verfahren Fuzzy-Vault [JS06] zählt zu den bekanntesten und meist akzeptierten Template Protection Verfahren und wurde für ungeordnete biometrische Merkmale unterschiedlicher Länge entwickelt. Aus diesem Grund eignet es sich zum Schutz der Minutien aus Fingerabdrücken. Beim Enrolment wird ein Polynom $p(\alpha)$ mit Grad k bestimmt, dessen Koeffizienten $Z = [a_0, \dots, a_k] : p(\alpha) = \sum_{i=0}^k a_i \cdot \alpha^i$ zufällig gewählt werden. Die Koordinaten (m_x, m_y) jeder Minutie des Fingers werden zusammen als ein Element α des endlichen Körpers \mathbb{F}_q dargestellt, wobei q so groß gewählt werden muss, dass alle Minutienpositionen eindeutig darin abgebildet werden können. Die Minutien-Informationen bilden zusammen mit den jeweiligen Funktionswerten $\beta = p(\alpha)$ die Stützpunkte des Polynoms. Zusätzlich werden sogenannte Chaff-Punkte generiert, die nicht auf p liegen. Diese dienen dazu, die echten Minutien zu verschleiern. Dazu werden für alle Chaff-Punkte zufällige $\beta \neq p(\alpha)$ generiert. Die Stützpunkte von p werden mit den Chaff-Punkten vermischt und als Datenbanktemplate R_j zusammen mit dem Hashwert $h(Z)$ gespeichert.

Verifikation. Bei der Verifikation werden die Minutien $m_Q \in Q$ mit denen des Referenztemplates R_j verglichen. Anhand konkreter globaler Translationen und Rotationen von Q , sowie einer Toleranz bzgl. Positionsabweichungen werden die übereinstimmenden Punkte ermittelt. Diese werden zusammen mit den entsprechenden Werten β_i aus R_j zur Rekonstruktion des Polynoms mit Hilfe des Reed-Solomon-Dekoders [RS60] verwendet, welches anschließend mit dem in der Datenbank hinterlegten Wert $h(Z)$ verglichen wird.

3 Effiziente Filter-Techniken

Für die Identifikation einer Person könnten naiv Authentisierungssysteme, wie beispielsweise das in [MIK⁺10] publizierte Verfahren, derart eingesetzt werden, dass sequentiell die gesamte Datenbank durchsucht wird bis ein Treffer erfolgt. Da diese Verfahren durch das explizite Austesten aller natürlichen Transformationen (Rotation, Translation) des Anfragefingers sehr ineffizient sind, ist dies jedoch für große Datenbanken nicht praktikabel. Stattdessen muss die Menge der zu überprüfenden Referenztemplates zuvor geeignet gefiltert werden, damit die exakte Verifikation nur auf einer reduzierten Menge von Personen durchgeführt wird. Die von uns vorgeschlagenen Verfahren dienen beide dazu, ein Ranking der Personen aufzustellen, absteigend sortiert nach Ähnlichkeit bezüglich der angefragten Person P_Q , wodurch die Zeit für den gesamten Identifikationsprozess verringert wird.

Um die Sicherheit vor Brute-Force Angriffen zu erhöhen, bietet es sich an, eine Person über mehrere Finger zu identifizieren. Eine Anfrage P_Q und das entsprechende Referenzobjekt einer Datenbank $P_j \in DB$ bestehen somit allgemein aus $\theta \in \{1, \dots, 10\}$ Templates: $P_Q = \{Q_1, \dots, Q_\theta\}$ und $P_j = \{R_{j,1}, \dots, R_{j,\theta}\}$. Ein Template der Anfrage $Q_f \in P_Q$ ist eine Menge von Minutenkoordinaten $m_Q = (m_x, m_y)$. Alle Referenztemplates $R_{j,f}$ einer Person P_j werden zusätzlich durch zufällig eingestreute Chaff-Punkte verschleiert. Da für jedes Template $R_{j,f}$ der zugehörige Fingertyp $f \leq \theta$ bekannt ist, kann angenommen werden, dass jeder Fingertyp in einem separaten Datenraum DB_f verwaltet wird.

3.1 GeoMatch

Der Vergleich zweier Templates Q_f und $R_{j,f}$ des entsprechenden Fingertyps f stellt durch das typischerweise vorliegende Rauschen und vor allem durch die Verschleierung eine Herausforderung dar. Ein einfacher Abgleich der Koordinaten beider Punktmengen ist hier nicht zielführend, stattdessen muss eine sehr große Menge von Transformationen in Betracht gezogen werden. Für diesen ersten Ansatz GeoMatch bedienen wir uns einiger Prinzipien aus dem verwandten Dockingproblem für Proteine. Ein typischer Ansatz hier ist es, das Problem in kleinere Elemente zu zerlegen, auf deren Basis individuelle Matchings durchgeführt werden. Anschließend werden diese lokalen Lösungen auf globale Konsistenz hin überprüft. Um einen Großteil der fraglichen Transformationen ausschließen zu können, wird beispielsweise in [Len95] ein Vergleich von Dreiecken, gebildet durch Zentren relevanter Moleküle, zu Grunde gelegt.

Für das vorliegende Problem des Vergleichs zweier Punktmengen Q_f und $R_{j,f}$ werden für beide Mengen Tripel berechnet, deren paarweise euklidischen Distanzen einen Schwellwert l_u übersteigen, sowie einen Schwellwert l_o unterschreiten. Die so gebildeten Dreiecke D_{Q_f} des Anfragetemplates werden mit den Dreiecken $D_{R_{j,f}}$ des Datenbanktemplates anhand der Seitenlängen (euklidische Distanz zweier Minuten) auf Ähnlichkeit getestet. Der Vergleich der lokalen Strukturen erfolgt somit unabhängig von Koordinatenwerten und vernachlässigt sowohl ihre globale Ausrichtung als auch ihre globale Positionierung. Um den Einfluss lokaler Positionsfehler der Minuten, begründet durch Ungenauigkeiten beim Enrolment oder bei der Minutenextraktion, zu schwächen, wird bei dem Vergleich der Seitenlängen eine Fehlertoleranz δ berücksichtigt. Sind zwei Dreiecke $d_a \in D_{Q_f}$ und $d_b \in D_{R_{j,f}}$ ähnlich, so wird ihre relative Rotation γ zueinander ermittelt. Für eine globale Prüfung auf Konsistenz wird im Anschluss überprüft, für wie viele Dreiecke in D_{Q_f} ein Dreieck in $D_{R_{j,f}}$ mit gleicher relativer Rotation vorliegt. Eine hohe Anzahl solcher Dreiecke lässt neben der vielen lokalen Matches auch Rückschlüsse auf eine globale Drehung ähnlicher Gesamtstrukturen zu. Je höher also die Anzahl ähnlicher lokaler Rotationen, desto wahrscheinlicher die Ähnlichkeit beider Templates. Die Ähnlichkeit zweier Templates Q_f und $R_{j,f}$ bestimmt sich somit wie folgt, wobei $\mathcal{A} = \{0^\circ, \dots, 360^\circ\}$ die Menge aller Winkel in einer zu wählenden Diskretisierung (z.B. 2° -Schritte) ist:

$$sim(D_{Q_f}, D_{R_{j,f}}) = \max_{\gamma \in \mathcal{A}} \left\{ \left| \{d_a \in D_{Q_f} \mid \exists d_b \in D_{R_{j,f}}. |d_a - d_b| \leq \delta \wedge \angle(d_a, d_b) \approx \gamma\} \right| \right\}$$

Für alle Referenztemplates $R_{j,f} \in DB_f$ wird die maximale Anzahl der Anfragedreiecke bestimmt, für die eine gleiche relative Rotation bzgl. Q_f ermittelt wurde. Dieser Prozess wird parallel für alle θ Finger der Anfrage Q durchgeführt. Für alle Personen $P_j \in DB$ werden anschließend die θ Ähnlichkeiten aller Finger $R_{j,i} \in P_j$ aufaddiert:

$$\text{sim}(P_Q, P_j) = \sum_{f \in \theta} \text{sim}(D_{Q_f}, D_{R_{j,f}})$$

Im Anschluss kann die Datenbank anhand dieser Anzahl absteigend sortiert und so für den Verifikationsprozess priorisiert werden.

Das Verfahren GeoMatch zeichnet sich durch seine Robustheit gegenüber globaler Verschiebung, sowie globaler Rotation der Templates aus, da lediglich Distanzen zwischen Minuten, nicht aber ihre Koordinaten verglichen werden.

3.2 BioSimJoin

Der Nachteil von GeoMatch ist die hohe Laufzeit, die der Abgleich mehrerer Dreieckssstrukturen zwischen Q und einem Referenztemplate R_j mit sich bringt. Aus diesem Grund werden bei BioSimJoin nicht die geometrischen Beziehungen zwischen Minuten und Chaff-Punkten betrachtet, sondern vielmehr Vergleiche der Punktmengen mittels Bereichsanfragen unterstützt durch eine Indexstruktur durchgeführt.

BioSimJoin speichert die Minuten bzw. Chaff-Punkte aller Personen $m_{R_{j,f}} \in R_{j,f}$ in einem Datenraum des entsprechenden Fingertyps f . Dabei werden die x - und y -Koordinaten aller Minuten bzw. Chaff-Punkte in einer Indexstruktur aus der Familie der R-Bäume [Gut84] organisiert. Diese hierarchischen Indexstrukturen, die ursprünglich zur Speicherung von hochdimensionalen Daten entwickelt wurden, eignen sich daher für die Verwaltung von biometrischen Punktdaten. Sie ermöglichen eine effiziente Beantwortung von Bereichsanfragen (d. h. Rechtecks- bzw. Intervall-Anfragen), und sind zudem dynamisch, d.h. durch effiziente Einfüge- bzw. Löschoptionen kann die Struktur bei Veränderung des Datenbestandes effizient aktualisiert werden.

Aufgrund der vorliegenden Rauscheffekte wie Rotation bzw. Translation zwischen der Anfrage Q_f und den Datenbanktemplates $R_{j,f}$ kann kein direkter Punktvergleich durchgeführt werden. Daher wird bei der Anfrage für jede Minute $m_{Q_f} \in Q_f$ eine Bereichsanfrage mit Radius r durchgeführt. Für jede dieser Minuten m_{Q_f} werden im Datenraum des entsprechenden Fingertyps f diejenigen Punkte (Minuten oder Chaff-Punkte) bestimmt, die sich innerhalb des Bereiches mit Radius r um m_{Q_f} befinden, d.h. deren euklidischer Abstand r nicht überschreitet. Die Information, ob es sich bei den Punkten um Minuten oder Chaff-Punkte handelt ist dabei nicht bekannt, lediglich welcher Person P_j sie zugeordnet sind. Falls eine Minute $m_{R_{j,f}}$ einer Person P_j in den Radius der Anfrageminute m_{Q_f} fällt, wird die Anzahl der Treffer für P_j um 1 erhöht. Die resultierende Kandidatenliste entspricht einer Liste an Personen P_j die absteigend nach Anzahl der Treffern für diese Person sortiert ist. Der algorithmische Ablauf von BioSimJoin ist in Algorithmus 1 zusammengefasst. Wie auch bei GeoMatch kann die Berechnung für verschiedene Finger-

typen parallelisiert erfolgen. Schließlich werden die Treffer aller Finger für jede Person aufsummiert.

Algorithm 1 $filter_{BioSim,Join}(Q_f, r)$

```

candidates = [( $P_1, 0$ ), ( $P_2, 0$ ), ..., ( $P_n, 0$ )]
 $DB_f = [(x_{1,f}, y_{1,f}, R_{1,f}), \dots, (x_{n,f}, y_{n,f}, R_{n,f})]$ 
for all minutia  $m_{Q_f}$  in  $Q_f$  do
  for all minutia  $m_{R_{j,f}}$  in  $DB_f$  do
    if  $dist(m_{Q_f}, m_{R_{j,f}}) \leq r$  then
       $candidates[j].increment()$ 
    end if
  end for
end for
return Kandidatenliste sortiert nach Anzahl der Treffer

```

4 Biometrische Datenbanken

Für die Evaluierung beider Verfahren verwenden wir die zwei Datenbanken für Fingerabdrucksbilder NIST SD14 [WGT⁺07] und FVC-2002 DB1 [MMJP09]. Während die Bilder der wesentlich größeren NIST-Datenbank durch Scans von Tintenabdrücken entstanden und somit sehr starkes Rauschen enthalten, wurden die Aufnahmen der für heutige Erkennungssysteme wesentlich repräsentativeren FVC-Datenbank direkt digital erfasst. Um ein möglichst praxisnahes Enrolment zu simulieren, haben wir uns an die in [MIK⁺10] beschriebenen Anforderungen an eine biometrische Datenbank gehalten. Gleichzeitig gewährleisten wir einen hohen Schutz der Daten vor bekannten Angriffen, wie in [MMT09] formuliert. Die Minutien wurden mittels des NIST Algorithmus mindtct [WGT⁺07] extrahiert, nach Qualität gefiltert und anschließend mit Hilfe des Fuzzy Vault [JS06] geschützt.

Multi-Finger. Die Verwendung mehrerer Finger pro Person für das Enrolment steigert exponentiell die Sicherheit vor Brute-Force Attacken. Aus diesem Grund identifizieren wir eine Person anhand von drei Fingern, wie in [MIK⁺10] empfohlen.

Feature-Selektion des Referenztemplates R_j . Um eine möglichst hohe Qualität der Referenztemplates zu garantieren und Ungenauigkeiten aus Scanvorgang sowie Minutienextraktion zu verringern, werden lediglich zuverlässige Minutien verwendet. Dazu werden, wie in [MIK⁺10] beschrieben, die Minutien ermittelt, die aus mehreren Aufnahmen eines Fingers extrahiert wurden. Diese werden zudem anhand ihres von mindtct ausgegebenen Qualitätswerts *rel* derart gefiltert, dass über alle drei Finger einer Person hinweg die besten 90 Minutien gewählt werden. Diese 90 Minutien werden anschließend durch insgesamt 112 zufällig eingestreute Chaff-Punkte verschleiert.

Diese Parameter gewährleisten ein Sicherheitslevel von 2⁷⁰ gegen Angriffe, die versuchen, „echte“ von „unechten“ Minutien, also Chaff-Punkten, unterscheiden zu können [MIK⁺10].

Feature-Selektion des Anfragetemplates Q . Da für Q in der Regel nur eine Aufnahme vorliegt, entfällt die Feature-Extraktion hinsichtlich korrespondierender Minutien meh-

rerer Aufnahmen. Stattdessen erfolgt die Filterung ausschließlich mittels des Qualitätskriteriums *rel*, wobei dieser jedoch einen Mindestwert von 0.25 übersteigen muss.

Die original Datenbank FVC-2002 DB1 enthielt ursprünglich jeweils acht Aufnahmen für 110 Finger. Nach allen Vorverarbeitungsschritten resultiert eine Datenbank, in der jeweils drei Finger zu insgesamt 27 Personen zusammengefasst wurden. Für diese Personen sind insgesamt 2.430 Minuten und 3.024 Chaff-Punkte gespeichert.

Die original NIST SD14 enthält zu 2.700 Personen jeweils zwei Aufnahmen für alle zehn Finger. Da die Qualität dieser Datenbank nicht an den heutigen Standard heranreicht, und einige Aufnahmen beispielsweise durch handschriftliche Bemerkungen stark verunreinigt sind, wurden nur die Bildpaare verwendet, die durch den BOZORTH Matchingalgorithmus [WGT⁺07] als Abdrücke des gleichen Fingers erkannt werden. Für die verbleibenden 2.365 Personen werden die drei Anfragefinger gemäß folgender Priorisierung gewählt: linker/rechter Zeigefinger, linker/rechter Mittelfinger, linker/rechter Ringfinger, linker/rechter Daumen und linker/rechter kleiner Finger. Das entspricht einer Reihenfolge von (7, 2, 8, 3, 9, 4, 6, 1, 0, 5) gemäß der Codierung des Fingertyps nach [WGT⁺07]. Insgesamt enthält der erstellte Datenbestand 212.850 Minuten und 264.880 Chaff-Punkte.

5 Experimente

Zunächst untersuchen wir die Parametereinstellung beider Verfahren sowohl anhand der Datenbank FVC-2002 DB1 als auch der Datenbank NIST SD14. Die so ermittelten optimalen Parameter werden hinterher zur Untersuchung der Effektivität, sowie der Effizienz der Verfahren auf beiden Datenbanken eingesetzt. Anschließend evaluieren wir die Robustheit beider Verfahren gegenüber gedrehten oder verschobenen Daten und untersuchen, inwieweit Insertionen bzw. Deletionen von Minuten die Ergebnisse beeinflussen. Alle Ergebnisse sind stets über alle Personen der entsprechenden Datenmenge gemittelt. Das bedeutet, dass in allen Experimenten jede Person ein Mal als Anfrage verwendet wird. Die Resultate entsprechen somit jeweils repräsentativen Durchschnittswerten.

Die Zeitmessungen wurden für jeden Finger parallelisiert auf folgenden Rechnern durchgeführt: Intel Dual Core Xeon 7120 M CPUs bzw. Intel XEON E5345 CPUs mit je 2.33 bis 3.0 GHz und 16 GB RAM. Alle Verfahren wurden mittels Java JDK 6.0 implementiert.

5.1 Parameterevaluierung

Die Seitenbeschränkungen für das Verfahren GeoMatch l_u und l_o wurden so gewählt, dass für nahezu alle Minuten aller Referenztemplates ein Dreieck ohne Chaff-Punkte konstruiert werden kann und gleichzeitig die Gesamtzahl aller Dreiecke möglichst gering ist. Diese Bedingungen erfüllt beispielsweise die gewählte Beschränkung der Seitenlänge der Dreiecke auf 14 – 80 Pixel. Für die Fehlertoleranz zeigte ein Wert von $\delta = 1$ Pixel die besten Ergebnisse.

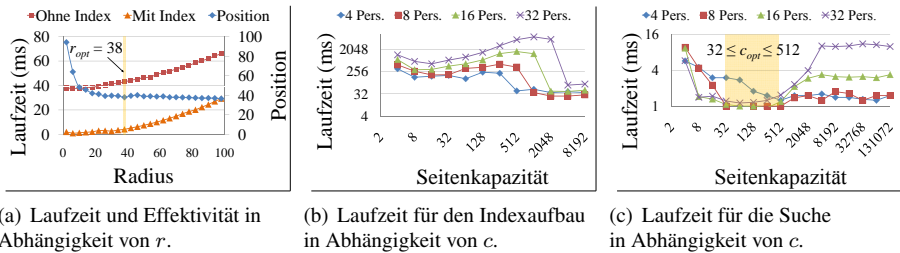


Abbildung 2: Bestimmung der optimalen Parameter von BioSimJoin.

Bei BioSimJoin müssen der Radius der Bereichsanfrage r und die maximale Kapazität einer Indexseite c geeignet gewählt werden. Da c lediglich die Effizienz des Verfahrens beeinflusst, kann zunächst eine Optimierung von r alleine durchgeführt werden. Abbildung 2(a) zeigt die entsprechenden Ergebnisse auf einem 10%igem Sample der Datenbank SD 14. Die durch Dreiecke und Quadrate markierten Kurven illustrieren die durchschnittlich benötigte Laufzeit, um die Kandidatenliste mit und ohne Indexunterstützung zu ermitteln. Ein höherer Radius impliziert die Überprüfung einer größeren Zahl von Datenbankelementen und daher eine erhöhte Laufzeit. Die mit Rauten markierte Kurve stellt die durchschnittliche Position der angefragten Person P_Q innerhalb der Kandidatenliste dar. Wenn r zu klein gewählt wird, kann P_Q erst relativ spät verifiziert werden. Den besten Kompromiss erzielt ein Radius $r_{opt} = 38$. BioSimJoin liefert nach durchschnittlich 6.23 ms bzw. 45.97 ms (ohne Indexunterstützung) eine Kandidatenliste, innerhalb derer sich P_Q durchschnittlich an Position 37.94 befindet. Unsere Experimente auf anderen biometrischen Datenbanken mit gleichem Setting ergaben, dass dieser Radius-Wert allgemein gute Ergebnisse verspricht.

Abbildung 2(b) zeigt den Zeitaufwand für die Indexierung, Abbildung 2(c) für die anschließende Suche, jeweils in Abhängigkeit von c . Diese Experimente wurden für unterschiedliche Datenbankgrößen durchgeführt. Die initialen Schwankungen sind durch Implementierungs-Overhead zu erklären. Für steigende Werte von c kann eine Zunahme der Laufzeit für den Indexaufbau beobachtet werden, da beim Splitten einer Seite sequenziell auf deren Elemente zugegriffen wird. Sobald c groß genug ist um alle Einträge in einer einzigen Seiten zu speichern, nimmt die Laufzeit der Indexierung rapide ab, wodurch allerdings auch keinerlei Indexunterstützung mehr für die anschließende Ähnlichkeitssuche gegeben ist. Diese Experimente lassen vermuten, dass eine optimale Kapazität $32 \leq c_{opt} < 512$, unabhängig von der Größe der angegebenen Datenbanken, sowohl für die Indexierung als auch die darauf aufbauende Suche gewählt werden sollte. Mittels eines sechsstufig gewichteten Mittelwerts der Laufzeiten für den relevanten Wertebereich $32 \leq c_{opt} < 512$ ergab sich ein globales Minimum bei $c_{opt} = 46$.

Diese Parametrisierungen beider Verfahren zeigten auf beiden Datenbanken gute Ergebnisse, sodass diese für alle folgenden Experimente übernommen wurden.

5.2 Effektivität

Für die FVC Datenbank gibt Tabelle 1 für GeoMatch sowie BioSimJoin jeweils die Position an, die eine angefragte Person P_Q gemittelt über alle 27 Anfragen in der Kandidatenliste einnimmt. Bei BioSimJoin wird die Kandidatenliste im Schnitt in 16.29 ms erzeugt, und P_Q ist auf Position 11.44 zu finden. Das Verfahren GeoMatch erzielt hier eine deutlich bessere Positionierung der korrekten Referenz in der Datenbank, benötigt für die entsprechenden Berechnungen allerdings signifikant mehr Zeit.

Abbildung 3 stellt das Ergebnis von GeoMatch und BioSimJoin anhand der Datenbank SD14 für unterschiedliche Datenbankgrößen dar. Trotz starkem Rauschen bei dieser Datenbank finden beide Verfahren die angefragte Person im Schnitt im vorderen Drittel der Kandidatenliste. Bei einer Datenbank bestehend aus 2.300 Personen kann mit GeoMatch die bei der anschließenden Verifikation zu überprüfende Anzahl an Personen um 69% reduziert werden. BioSimJoin schließt für die Verifikation 66% der Personen aus.

	GEOMATCH	BIOSIMJOIN
<i>Position</i>	2.07	11.44
<i>Laufzeit</i>	91.19 ms	16.29 ms

Tabelle 1: Effektivität bzgl. Datenbank FVC.

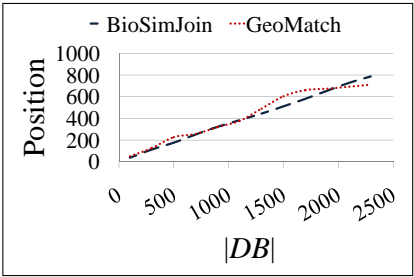


Abbildung 3: Effektivität bzgl. Datenbank SD14.

5.3 Effizienz

Abbildung 4 untersucht die Skalierbarkeit von GeoMatch und BioSimJoin anhand der Datenbank SD14. GeoMatch zeigt hier eine lineare Laufzeit mit zunehmender Anzahl an Datenbankeinträgen (Minutien bzw. Chaff-Punkten). Durch den Einsatz einer Indexstruktur erzielt BioSimJoin eine deutlich geringere Laufzeit. So benötigt BioSimJoin für eine Anfrage auf eine Datenbank bestehend aus knapp 500.000 Einträgen lediglich 2.4 Sekunden, GeoMatch hingegen 2 Minuten und 10 Sekunden.

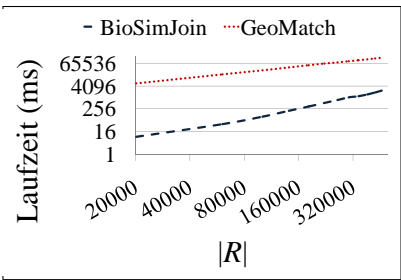


Abbildung 4: Skalierbarkeit der Verfahren.

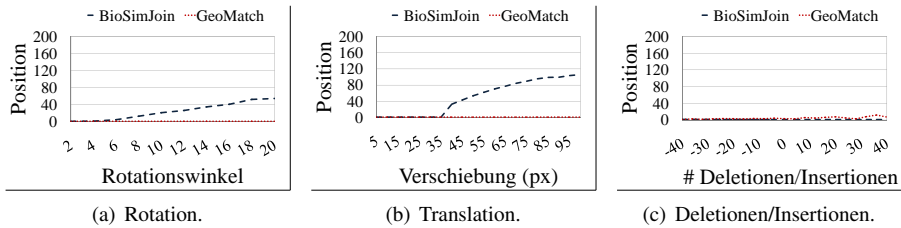


Abbildung 5: Robustheit von GeoMatch und BioSimJoin gegenüber unterschiedlichem Rauschen.

5.4 Evaluierung der Robustheit anhand synthetischer Daten

Um die Stabilität von GeoMatch und BioSimJoin gegenüber Rotation, Translation, fehlen oder zusätzlichen Minutien zu testen, wurden jeweils die Minutien von 200 zufällig ausgewählten Personen der Referenzdatenbank SD14 gezielt manipuliert und als Anfrage verwendet. Für jedes Experiment ist die Position der angefragten Person P_Q innerhalb der Kandidatenliste gemittelt über 200 Anfragen angegeben.

Rotation. Um die Auswirkung eines rotierten Anfragetemplates Q auf die Effektivität der Verfahren zu untersuchen, drehen wir das Koordinatensystem von Q um einen Rotationswinkel ϕ im Intervall $[2^\circ, 4^\circ, 6^\circ, \dots, 20^\circ]$. In der Regel wird beim Enrolment lediglich eine Rotation um bis zu 20° toleriert. Minutien, die durch die Rotation aus dem Bildbereich fallen, wurden verworfen. Abbildung 5(a) zeigt, dass die Effektivität von BioSimJoin mit zunehmender Drehung der Minutien aus Q leicht abnimmt, da die Minutien der Referenz nicht mehr optimal in die entsprechenden Anfrageradien fallen. Allerdings liegt P_Q bei einer starken Drehung von 20° im Mittel in der Kandidatenliste immer noch auf einem guten Platz 54 von insgesamt 200, obwohl die Drehung bei BioSimJoin nicht explizit berücksichtigt wird. Das Verfahren GeoMatch ist robust gegen Rotationen.

Translation. Um abweichende Auflagepositionen des Anfragefingers auf dem Scanner zu simulieren, wurden die Minutien-Koordinaten aus Q einheitlich um jeweils x Pixel verschoben. Auch hier wurden verschobene Minutien außerhalb des Bildbereichs ausgeschlossen. Bei einer Verschiebung um einen Wert kleiner dem Radius r der Bereichsanfrage von BioSimJoin bleiben die gesuchten Personen innerhalb der Kandidatenliste stabil auf dem ersten Platz (vgl. Abbildung 5(b)). Erst bei einer darüber hinausgehenden Verschiebung, fallen einige Minutien der Referenz aus der Bereichsanfrage, sodass die Effektivität leicht abnimmt. GeoMatch ist hier robust und zeigt konstant optimale Ergebnisse.

Insertionen und Deletionen. Bedingt durch Ungenauigkeiten im Scanvorgang oder der Minutienextraktion werden für Q teilweise andere Minutien erkannt als für die Referenz. Für entsprechende Untersuchungen wurden Q im Vergleich zu den Referenzdaten zufällig Minutien hinzugefügt bzw. entfernt. Abbildung 5(c) zeigt, dass für das Verfahren BioSimJoin das Fehlen bzw. Hinzukommen von bis zu 40 Minutien keine Auswirkung auf die optimale Erkennungsleistung haben. Die Ergebnisse von GeoMatch unterliegen hingegen leichten Schwankungen. Besonders bei einer großen Anzahl zusätzlicher Minutien kommt es vor, dass einige der zusätzlichen Minutien im Anfragetemplate auf Chaff-Punkte fremder Referenztemplates matchen, wodurch diese fälschlich im Ranking begünstigt werden.

6 Zusammenfassung

Wir haben erste Ansätze vorgestellt, die eine Personenidentifikation anhand ihres geschützten Fingerabdrucks effizient ermöglichen. Bisherige Verfahren unterstützen lediglich eine Identifikation auf ungeschützten Daten oder eine Authentifikation auf sehr kleinen Datenmengen. Die von uns entwickelten Filtertechniken erzeugen ein priorisiertes Ranking, anhand dessen ein genauer Abgleich von Anfrage und geschütztem Referenzobjekt durchgeführt wird. Experimente auf realen und synthetischen Daten zeigen, dass trotz starkem Rauschen wie Rotation, Translation und Insertionen bzw. Deletionen bei der Anfrage, und der Verschleierung der Referenzdaten, eine effektive und effiziente Identifikation ermöglicht wird. Während sich das Verfahren GeoMatch hauptsächlich durch Rotations- und Translationsinvarianz auszeichnet, werden starke Effizienzsteigerungen in erster Linie durch das zweite Verfahren BioSimJoin erzielt. In naher Zukunft werden wir uns damit beschäftigen, beide Kriterien in ein Verfahren zu integrieren.

Danksagung: Diese Arbeit wurde innerhalb des Projekts BioKeyS des Bundesamt für Sicherheit in der Informationstechnik (BSI) durch den Zukunftsfond gefördert. Wir danken allen Partnern insbesondere Sebastian Abt, Christoph Busch, Heinrich Ihmor, Claudia Nickel, Alexander Nouak, Alexander Opel und Xuebing Zhou für die erfolgreichen Diskussionen und zahlreichen Kommentare.

Literatur

- [Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD Conference*, Seiten 47–57, 1984.
- [JS06] A. Juels und M. Sudan. A Fuzzy Vault Scheme. *Des. Codes Cryptography*, 38(2):237–257, 2006.
- [KMN09] U. Korte, J. Merkle und M. Niesing. Datenschutzfreundliche Authentisierung mit Fingerabdrücken. *Datenschutz und Datensicherheit - DuD*, 33(5):289–294, May 2009.
- [Len95] H.-P. Lenhof. An Algorithm for the Protein Docking Problem. In *Bioinformatics: From Nucleic Acids and Proteins to Cell Metabolism*, Seiten 125–139, 1995.
- [MIK⁺10] J. Merkle, H. Ihmor, U. Korte, M. Niesing und M. Schwaiger. Performance of the Fuzzy Vault for Multiple Fingerprints (Extended Version). *CoRR*, abs/1008.0807, 2010.
- [MMJP09] D. Maltoni, D. Maio, A.K. Jain und S. Prabhakar. *Handbook of Fingerprint Recognition*. Springer Publishing Company, Incorporated, 2009.
- [MMT09] P. Mihalescu, A. Munk und B. Tams. The Fuzzy Vault for Fingerprints is Vulnerable to Brute Force Attack. In *BIOSIG*, Seiten 43–54, 2009.
- [RS60] I. S. Reed und G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
- [WGT⁺07] C.I. Watson, M.D. Garris, E. Tabassi, C.L. Wilson, R.M. McCabe, S. Janet und K. Ko. User's Guide to NIST Biometric Image Software (NBIS), National Institute of Standards and Technology, 2007.

Benchmarking Hybrid OLTP&OLAP Database Systems

Florian Funke

Alfons Kemper
{first.last}@in.tum.de

Thomas Neumann

Technische Universität München
Fakultät für Informatik
Boltzmannstr. 3
85748 Garching, Germany

Abstract: Recently, the case has been made for operational or real-time Business Intelligence (BI). As the traditional separation into OLTP database and OLAP data warehouse obviously incurs severe latency disadvantages for operational BI, hybrid OLTP&OLAP database systems are being developed. The advent of the first generation of such hybrid OLTP&OLAP database systems requires means to characterize their performance.

While there are standardized and widely used benchmarks addressing either OLTP or OLAP workloads, the lack of a hybrid benchmark led us to the definition of a new mixed workload benchmark, called TPC-CH. This benchmark bridges the gap between the existing single-workload suits: TPC-C for OLTP and TPC-H for OLAP. The newly proposed TPC-CH benchmark executes a mixed workload: A transactional workload based on the order entry processing of TPC-C and a corresponding TPC-H-equivalent OLAP query suite on this sales data base. As it is derived from these two most widely used TPC benchmarks our new TPC-CH benchmark produces results that are highly comparable to both, hybrid systems and classic single-workload systems. Thus, we are able to compare the performance of our own (and other) hybrid database system running both OLTP and OLAP workloads in parallel with the OLTP performance of dedicated transactional systems (e.g., VoltDB) and the OLAP performance of specialized OLAP databases (e.g., column stores such as MonetDB).

1 Introduction

The two areas of online transactions processing (OLTP) and online analytical processing (OLAP) constitute different challenges for database architectures. While transactions are typically short-running and perform very selective data access, analytical queries are generally longer-running and often scan significant portions of the data. Therefore customers with high rates of mission-critical transactions are currently forced to operate two separate systems: one operational database processing transactions and one *data warehouse* dedicated to analytical queries. The data warehouse is periodically updated with data that is extracted from the OLTP system and transformed into a schema optimized for analysis. Early attempts to run analyses directly on the operational systems resulted in unacceptable transaction processing performance [DHKK97].

While this data staging approach allows each system to be tuned for its respective workload, it suffers from several inherent drawbacks: Two software and hardware systems must be purchased and maintained. Additional systems may be required depending on the data staging implementation. All systems have to store redundant copies of the same data, but most importantly, analyses do not incorporate the latest data, but work on the stale snapshot in the data warehouse.

Recently, the case has been made for so called *real-time Business Intelligence*. SAP's co-founder Hasso Plattner [Pla09] criticizes the separation between OLTP and OLAP deploring a shift of priorities towards OLTP. He emphasizes the necessity of OLAP for strategic management and compares the expected impact of real-time analysis on management with the impact of Internet search engines on the world.

Real-time business intelligence postulates novel types of database architectures, often based on in-memory technology, such as the Hybrid Row-Column OLTP Database Architecture for Operational Reporting [SBKZ08, BHF09, KGT⁺10] or HyPer [KN11]. They address both workloads with a single system, thereby eliminating the aforementioned shortcomings of the data staging approach.

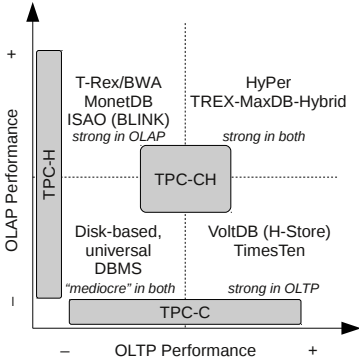


Figure 1: Classification of DBMS and Benchmarks

Different strategies seem feasible to reconcile frequent inserts and updates with longer running BI queries: Modifications triggered by transactions may be collected in a delta and periodically merged with the main dataset which serves as a basis for queries [KGT⁺10]. Alternatively, the DBMS can devise versioning to separate transaction processing on the latest version from queries operating on a snapshot of the versionized data.

This novel class of DBMS necessitates means to analyze their performance. Hybrid systems need to be compared against each other to evaluate the different implementation strategies. They must also be juxtaposed to traditional, universal DBMS and specialized single-workload systems to prove their competitiveness in terms of performance and resource consumption.

We present TPC-CH, a benchmark that seeks to produce highly comparable results for all types of systems (cf. Figure 1). The following section evaluates related benchmarks. Section 3 describes the design of the TPC-CH. Section 4 describes the systems under test.

Section 5 shows setups and results produced with different types of DBMS and Section 6 concludes the paper.

2 Related Work

The Transaction Processing Performance Council (TPC) specifies benchmarks that are widely used in industry and academia to measure performance characteristics of database systems. TPC-C and its successor TPC-E simulate OLTP workloads. The TPC-C schema consists of nine relations and five transactions that are centered around the management, sale and distribution of products or services. The database is initially populated with random data and then updated as new orders are processed by the system. TPC-E simulates the workload of a brokerage firm. It features a more complex schema and pseudo-real content that seeks to match actual customer data better. However, TPC-C is far more pervasive compared to TPC-E [Tra10c, Tra10d] and thus offers better comparability.

TPC-H is currently the only active decision support benchmark of the TPC. It simulates an analytical workload in a business scenario similar to TPC-C's. The benchmark specifies 22 queries on the 8 relations that answer business questions. TPC-DS, its dedicated successor, will feature a star-schema, around 100 decision support queries and a description of the ETL process that populates the database. However, it currently is in draft state.

Note that composing a benchmark for hybrid DBMS by simply using two TPC schemas, one for OLTP and one for OLAP, does not produce meaningful results. Such a benchmark would not give insight into how a system handles its most challenging task: The concurrent processing of transactions and queries on the *same* data.

The composite benchmark for online transaction processing (CBTR) [BKS08] was proposed to measure the impact of a workload that comprises both OLTP and operational reporting. CBTR is no combination of existing standardized benchmarks, but uses an enterprises' real data. The authors mention the idea of a data generator to produce results that allow comparisons between systems. Yet the focus of CBTR seems to be the comparison of different database systems for the specific use case of a certain enterprise.

3 Benchmark Design

Our premier goal in the design of TPC-CH was comparability. Therefore, we leverage a combination of TPC-C and TPC-H. Both benchmarks are widely used and accepted, relatively fast to implement and have enough similarity in their design to make a combination possible.

TPC-CH is comprised of the unmodified TPC-C schema and transactions and an adapted version of the TPC-H queries. Since the schemas of both benchmarks (cf. Figure 2) model businesses which “must manage, sell, or distribute a product or service” [Tra10a, Tra10b], they have some similarities between them. The relations ORDER(S) and CUSTOMER exist

in both schemas. Moreover, both ORDER-LINE (TPC-C) and LINEITEM (TPC-H) model entities that are sub-entities of ORDER(S) and thus resemble each other.

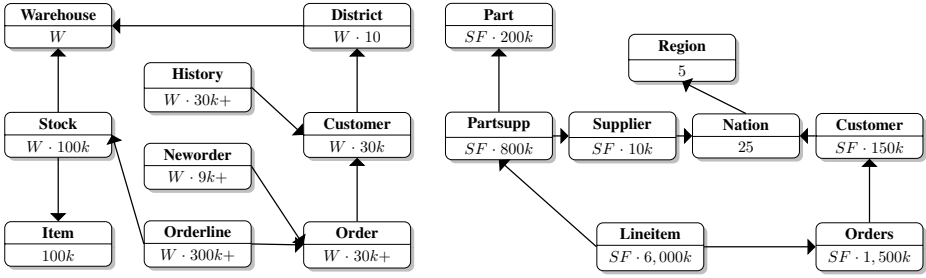


Figure 2: TPC-C and TPC-H schemas

TPC-CH keeps all TPC-C entities and relationships completely unmodified and integrates the likewise unchanged relations SUPPLIER, REGION and NATION from the TPC-H schema. These relations are frequently used in TPC-H queries and allow a non-intrusive integration into the TPC-C schema. The relation SUPPLIER is populated with a fixed number (10,000) of entries. Thereby, an entry in STOCK can be uniquely associated with its SUPPLIER through the relationship $STOCK.S_I_ID \times STOCK.S_W_ID \bmod 10,000 = SUPPLIER.SU_SUPPKEY$.

The original TPC-C relation CUSTOMER contains no foreign key that references the associated NATION. Since we keep the original schema untouched to preserve compatibility with existing TPC-C installations, the foreign key is computed from the first character of the field C_STATE. TPC-C specifies that this first character can have 62 different values (upper-case letters, lower-case letters and numbers), therefore we chose 62 nations to populate NATION. The primary key N_NATIONKEY is an identifier according to the TPC-H specification. Its values are chosen such that their associated ASCII value is a letter or number (i.e. $N_NATIONKEY \in [48, 57] \cup [65, 90] \cup [97, 122]$). Therefore no additional calculations are required to skip over the gaps in the ASCII code between numbers, upper-case letters and lower-case letters. Database systems that do not provide a conversion routine from a character to its ASCII code may deviate from the TPC-H schema and use a single character as a primary key for NATION. REGION contains the five regions of these nations. Relationships between the new relations are modeled with simple foreign key fields (NATION.N_REGIONKEY and SUPPLIER.SU_NATIONKEY).

3.1 Transactions and Queries

As illustrated in the overview in Figure 4, the workload consists of the five original TPC-C transactions and 22 queries adopted from TPC-H. Since the TPC-C schema is an unmodified subset of the TPC-CH schema, the original transactions can be executed without any modification:

New-Order This transaction enters an order with multiple order-lines into the database. For each order-line, 99% of the time the supplying warehouse is the home ware-

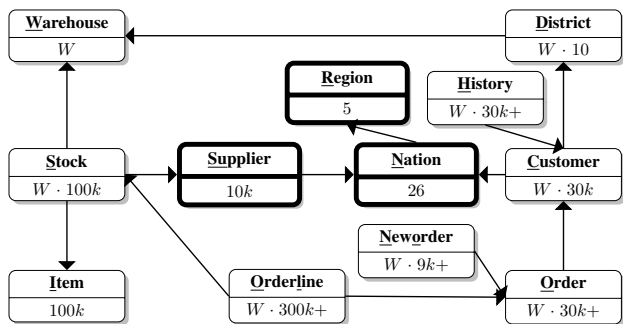


Figure 3: TPC-CH schema. Entities originating from TPC-H are highlighted.

house. The home warehouse is a fixed warehouse ID associated with a terminal. To simulate user data entry errors, 1% of the transactions fail and trigger a roll-back.

Payment A payment updates the balance information of a customer. 15% of the time, a customer is selected from a random remote warehouse, in the remaining 85%, the customer is associated with the home warehouse. The customer is selected by last name in 60% of the cases and else by his three-component key.

Order-Status This read-only transaction is reporting the status of a customer's last order. The customer is selected by last name 60% of the time. If not selected by last name, he is selected by his ID. The selected customer is always associated with the home warehouse.

Delivery This transaction delivers 10 orders in one batch. All orders are associated with the home warehouse.

Stock-Level This read-only transaction operates on the home warehouse only and returns the number of those stock items that were recently sold and have a stock level lower than a threshold value.

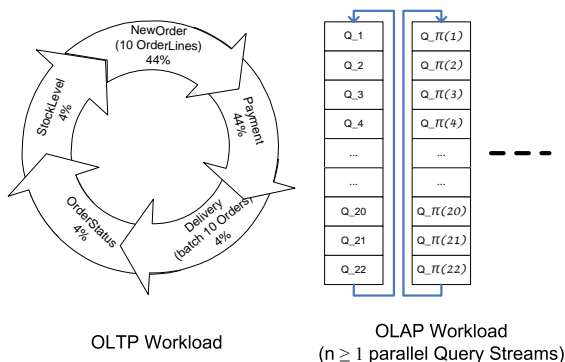


Figure 4: Benchmark overview: OLTP and OLAP on the same data

The distribution over the five transaction types conforms to the TPC-C specification (cf. Figure 4), resulting in frequent execution of the New-Order and Payment transactions.

TPC-CH deviates from the underlying TPC-C benchmark by not simulating the terminals and by generating client requests without any think-time, as proposed by [Vola]. This way, very high transaction rates can be achieved on relatively small database configurations. Since the transactions themselves remain the same as in TPC-C, TPC-CH results are directly comparable to existing TPC-C results with the same modifications, e.g. VoltDB [Vola]. Moreover, these changes can be easily applied to existing TPC-C implementations in order to produce TPC-CH-compatible results.

For the OLAP portion of the workload, we adopt the 22 queries from TPC-H to the TPC-CH schema. In reformulating the queries to match the slightly different schema, we made sure that their business semantics and syntactical structure were preserved. E.g., query 5 lists the revenue achieved through local suppliers (cf. Listing 1 and 2). Both queries join similar relations, have similar selection criteria, perform summation, grouping and sorting.

```
SELECT n_name, SUM(ol_amount) AS revenue
FROM customer, "order", orderline, stock, supplier, nation, region
WHERE c_id=o_c_id AND c_w_id=o_w_id AND c_d_id=o_d_id
      AND ol_o_id=o_id AND ol_w_id=o_w_id AND ol_d_id=o_d_id
      AND ol_w_id=s_w_id AND ol_i_id=s_i_id
      AND mod((s_w_id * s_i_id),10000)=su_suppkey
      AND ascii(SUBSTRING(c_state, 1, 1))=su_nationkey
      AND su_nationkey=n_nationkey AND n_regionkey=r_regionkey
      AND r_name=' [REGION]' AND o_entry_d>=' [DATE]'
GROUP BY n_name ORDER BY revenue DESC
```

Listing 1: TPC-CH Query 5

```
SELECT n_name, SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM customer, orders, lineitem, supplier, nation, region
WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey
      AND l_suppkey = s_suppkey AND c_nationkey = s_nationkey
      AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey
      AND r_name = ' [REGION]' AND o_orderdate >= DATE ' [DATE]'
      AND o_orderdate < DATE ' [DATE]' + INTERVAL '1' YEAR
GROUP BY n_name ORDER BY revenue DESC
```

Listing 2: TPC-H Query 5

TPC-CH does not require refresh functions, as specified in TPC-H, since the TPC-C transactions are continuously updating the database. The following section particularizes, when queries have to incorporate these updates.

3.2 Benchmark Parameters

TPC-CH has four scales: First, the database size is variable. As in TPC-C, the size of the database is specified through the number of warehouses. Most relations grow with the number of warehouses, with Item, Supplier, Nation and Region being the only ones of constant size.

The second scale is the composition of the workload. It can be comprised of analytical queries only, transactions only or any combination of the two. The workload mix is specified as the number of parallel OLTP and OLAP sessions (streams) that are connected to the database system. An OLTP session dispatches random TPC-C transactions sequentially with the distribution described in the official specification [Tra10a]. An analytical session performs continuous iterations over the query set which is comprised of all 22 queries. Each session starts with a different query to avoid caching effects between sessions as depicted in Figure 4.

The third input parameter is the isolation level. Lower isolation levels like read committed allow for faster processing, while higher isolation levels guarantee higher quality results for both transactions and queries.

Finally, the freshness of the data that is used as a basis for the analyses is a parameter of the benchmark. It only applies if the workload mix contains both, OLTP and OLAP components. Data freshness is specified as the time or the number of transactions after which newly issued query sets have to incorporate the most recent data. This allows for both, database architectures that have a single dataset for both workloads and those that devise a delta to run the benchmark.

3.3 Reporting requirements

In addition to a description of the hard- and software employed, the following characteristics of the system are reported: The OLTP engine’s performance is quantified by the throughput of New-Order transactions and all transactions. On the OLAP side, the query response time is measured for each query in each iteration and session. The median value and the query throughput are reported. In addition to the freshness parameter, the maximum dataset age is reported. For in-memory systems, the total memory consumption of all processes over time is reported. This includes allocated, but not yet used memory chunks. Figure 5 shows a sample report.

OLTP		configuration	
# threads		OLAP	
isolation level		# query sessions (streams)	
		isolation method	
new order: # tps total: # tps	query throughput	Q1: median resp. time	
	max dataset age of any query	Q2: median resp. time	
		...	
		Q22: median resp. time	

Figure 5: Reporting Requirements of the TPC-CH Benchmark

4 Systems under Test

In Figure 1, we grouped DBMSs in four segments. We use TPC-CH to analyze the performance of one representative of each category.

4.1 OLAP-focused Database Systems

MonetDB is the most influential database research project on column store storage schemes for in-memory OLAP databases. An overview of the system can be found in the summary paper [BMK09] presented on the occasion of receiving the 10 year test of time award of the VLDB conference. Therefore, we use MonetDB as a representative of the “strong in OLAP”-category. Other systems in this category are TREX (BWA) of SAP, IBM Smart Analytics Optimizer and Vertica Analytic Database.

4.2 OLTP-focused Database Systems

The H-Store prototype [KKN⁺08], created by researchers led by Michael Stonebraker was recently commercialized by a start-up company name VoltDB. VoltDB is a high-performance, in-memory OLTP system that pursues a lock-less approach [HAMS08] to transaction processing where transactions operate on private partitions and are executed in serial [Volb]. VoltDB represents the “strong in OLTP”-category. This category also includes the following systems: P*Time [CS04], IBM solidDB, TimesTen of Oracle and the new startup developments Electron DB, Clustrix, Akiban, dbShards, NimbusDB, ScaleDB and Lightwolf.

4.3 Universal Database Systems

This category contains the disk-based, universal database systems. We picked a popular, commercially available one (“System X”) as a representative of the universal DBMS category.

4.4 Hybrid Database Systems

This category includes the new database development at SAP as outlined by Hasso Plattner [Pla09] and our HyPer [KN11] system. A special-purpose OLTP&OLAP system is Crescendo [GUMG10] that has, however, limited query capabilities.

4.4.1 HyPer: Virtual Memory Snapshots

We have developed a novel hybrid OLTP&OLAP database system based on snapshotting transactional data via the virtual memory management of the operating system [KN11]. In this architecture the OLTP process “owns” the database and periodically (e.g., in the order of seconds or minutes) forks an OLAP process. This OLAP process constitutes a fresh transaction consistent snapshot of the database. Thereby, we exploit the operating

systems functionality to create virtual memory snapshots for new, duplicated processes. In Unix, for example, this is done by creating a child process of the OLTP process via the `fork()` system call. To guarantee transactional consistency, the `fork()` should only be executed in between two (serial) transactions, never in the middle of one transaction. In section 4.4.1 we will relax this constraint by utilizing the undo log to convert an action consistent snapshot (created in the middle of a transaction) into a transaction consistent one.

The forked child process obtains an exact copy of the parent processes address space, as exemplified in Figure 6 by the overlaid page frame panel. This virtual memory snapshot that is created by the `fork()`-operation will be used for executing a session of OLAP queries – as indicated on the right hand side of Figure 6.

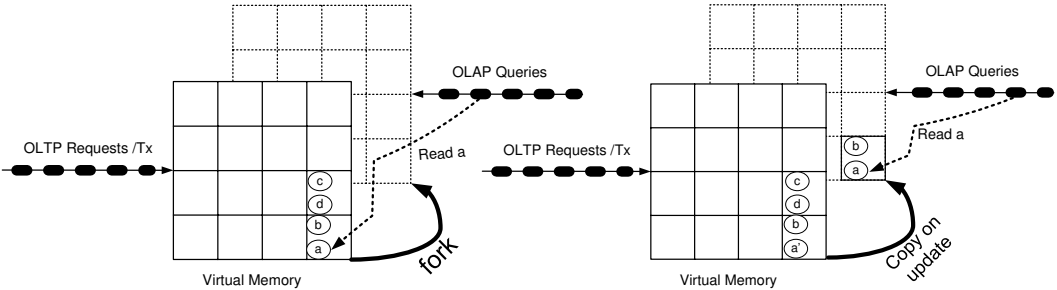


Figure 6: Forking a new Snapshot (left) and copy-on-update/write (right)

The snapshot stays in precisely the state that existed at the time the `fork()` took place. Fortunately, state-of-the-art operating systems do not physically copy the memory segments right away. Rather, they employ a lazy *copy-on-update* strategy – as sketched out in Figure 6. Initially, parent process (OLTP) and child process (OLAP) share the same physical memory segments by translating either virtual addresses (e.g., to object *a*) to the same physical main memory location. The sharing of the memory segments is highlighted in the graphics by the dotted frames. A dotted frame represents a virtual memory page that was not (yet) replicated. Only when an object, like data item *a*, is updated, the OS- and hardware-supported copy-on-update mechanism initiate the replication of the virtual memory page on which *a* resides. Thereafter, there is a new state denoted *a'* accessible by the OLTP-process that executes the transactions and the old state denoted *a*, that is accessible by the OLAP query session. Unlike the figure suggests, the additional page is really created for the OLTP process that initiated the page change and the OLAP snapshot refers to the old page – this detail is important for estimating the space consumption if several such snapshots are created (cf. Figure 7).

So far we have sketched a database architecture utilizing two processes, one for OLTP and another one for OLAP. As the OLAP queries are *read-only* they could easily be executed in parallel in multiple threads that share the same address space. Still, we can avoid any synchronization (locking and latching) overhead as the OLAP queries do not share any mutable data structures. Modern multi-core computers which typically have more than ten cores can certainly yield a substantial speed up via this inter-query parallelization.

Another possibility to make good use of the multi-core servers is to create multiple snapshots. The HyPer architecture allows for arbitrarily current snapshots. This can simply be

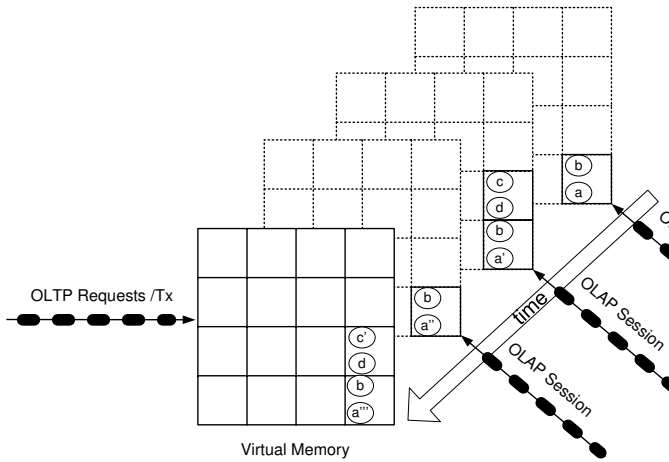


Figure 7: Multiple OLAP Sessions at Different Points in Time

achieved by periodically (or on demand) `fork()`-ing a new snapshot and thus starting a new OLAP query session process. This is exemplified in Figure 7. Here we sketch the one and only OLTP process's current database state (the front panel) and three active query session processes' snapshots – the oldest being the one in the background. The successive state changes are highlighted by the four different states of data item *a* (the oldest state), *a'*, *a''*, and *a'''* (the youngest transaction consistent state). Obviously, most data items do not change in between different snapshots as we expect to create snapshots for most up-to-date querying at intervals of a few seconds – rather than minutes or hours as is the case in current separated data warehouse solutions with ETL data staging. The number of active snapshots is, in principle, not limited, as each “lives” in its own process. By adjusting the priority we can make sure that the mission critical OLTP process is always allocated a core – even if the OLAP processes are numerous and/or utilize multi-threading and thus exceed the number of cores.

A snapshot will be deleted after the last query of a session is finished. This is done by simply terminating the process that was executing the query session. It is not necessary to delete snapshots in the same order as they were created. Some snapshots may persist for a longer duration, e.g., for detailed stocktaking purposes. However, the memory overhead of a snapshot is proportional to the number of transactions being executed from creation of this snapshot to the time of the next younger snapshot (if it exists or to the current time). The figure exemplifies this on the data item *c* which is physically replicated for the “middle age” snapshot and thus shared and accessible by the oldest snapshot. Somewhat against our intuition, it is still possible to terminate the middle-aged snapshot before the oldest snapshot as the page on which *c* resides will be automatically detected by the OS-/processor as being shared with the oldest snapshot via a reference counter associated with the physical page. Thus it survives the termination of the middle-aged snapshot – unlike the page on which *a'* resides which is freed upon termination of the middle-aged snapshot process. The youngest snapshot accesses the state *c'* that is contained in the current OLTP process's address space.

5 Results

In this section, we present cursory results with TPC-CH. Our experiments are performed on a machine with two quad-core 2.93 GHz Intel® Xeon® processors and 64GB of memory running Red Hat Enterprise Linux 5.4. All databases are scaled to 12 warehouses and we performed 5 iterations over the query sets.

For MonetDB, we evaluate an instance of the benchmark that performs a pure-OLAP workload. We excluded OLTP because the absence of indexes in MonetDB prevents efficient transaction processing. We present results of setups with three parallel OLAP sessions in Figure 9. Since there are no updates to the database in this scenario, the freshness and the isolation level parameter are lapsed. Increasing the number of query streams to 5 hardly changes the throughput, but almost doubled the query execution times. Running a single query session improved the execution times between 10% and 45% but throughput deteriorates to 0.55 queries per second.

For VoltDB, the workload-mix includes transactions only. One “site” per warehouse/partition (i.e. 12 sites) yields best results on our server. Differing from the TPC-CH specification, we allow VoltDB to execute only single-partition transactions, as suggested in [Vola] and skip those instances of New-Order and Payment that involve more than one warehouse. The isolation level in VoltDB is serializable.

For System X, we use 25 OLTP sessions and 3 OLAP sessions. The configured isolation level is read committed for both OLTP and OLAP and we use group committing with groups of five transactions. Since the system operates on a single dataset, every query operates on the latest data. Figure 9 shows the results of this setup. Increasing the OLAP sessions from 3 to 12 enhances the query throughput from 0.38 to 1.20 queries/s, but causes the query execution times to go up by 20 to 30% and a decline of the OLTP throughput by 14%. Adding more OLTP sessions drastically increased query execution time as well.

For HyPer, we use a transaction mix of 5 OLTP sessions and 3 parallel OLAP sessions executing queries. We do not make the simplification of running single-partition transactions only, as for VoltDB, but challenge HyPer with warehouse-crossing transactions as specified in Section 3.1. In one setup, the OLAP sessions operate on the initially loaded data (cf. Figure 9). In a second one, a fresh snapshot is created for every new query stream (cf. Figure 9). Queries are snapshot-isolated from transactions. On the OLTP side, the isolation level is serializable.

Since HyPer does not feature separate client and server processes yet, the results are produced by a single driver that incorporates both components. Thus, potential performance loss caused by inter-process communication is ruled out for HyPer, but not for the others systems under test. HyPer’s strong OLTP performance results from the compilation of transactions to machine code. VoltDB uses stored procedures written in Java instead.

Figure 9 shows the memory consumption of HyPer and VoltDB. We do not include MonetDB results here, because the MonetDB database does not grow over time. HyPer runs three query sessions concurrently to the 5 OLTP sessions and spawns a fresh VM snapshot after each iteration. VoltDB executes a pure-OLTP workload. The figure shows the memory consumption after the initial load was performed.

Query	System X		HyPer configurations		MonetDB no OLTP 3 query streams Query resp. times (ms)	VoltDB no OLAP only OLTP
	3 query streams 25 JDBC clients OLTP throughput	Query resp. times (ms)	8 query sessions (streams) single threaded OLTP OLTP throughput	3 query sessions (streams) 5 OLTP threads OLTP throughput		
Q1	new order: 221.91 tps; total: 493.14 tps	4221	new order: 25166 tps; total: 55924 tps	new order: 112217 tps; total: 249237 tps	72	new order: 16273.80 tps; total: 36159.07 tps According to [Volb]: 53000 tps (total) on one node; 560000 tps (total) on 12 nodes
Q2		6555			218	
Q3		16410			112	
Q4		3830			8168	
Q5		15212			12028	
Q6		3895			163	
Q7		8285			2400	
Q8		1655			306	
Q9		3520			214	
Q10		15309			9239	
Q11		6006			42	
Q12		5689			214	
Q13		918			521	
Q14		6096			919	
Q15		6768			587	
Q16		6088			7703	
Q17		5195			335	
Q18		14530			2917	
Q19		4417			4049	
Q20		3751			937	
Q21		9382			332	
Q22		8821			167	
Throughput		0.38 queries/s	10.49 queries/s	5.96 queries/s	1.21 queries/s	

Figure 8: Performance Comparison: System X, HyPer OLTP&OLAP, MonetDB (OLAP only), VoltDB (OLTP only)

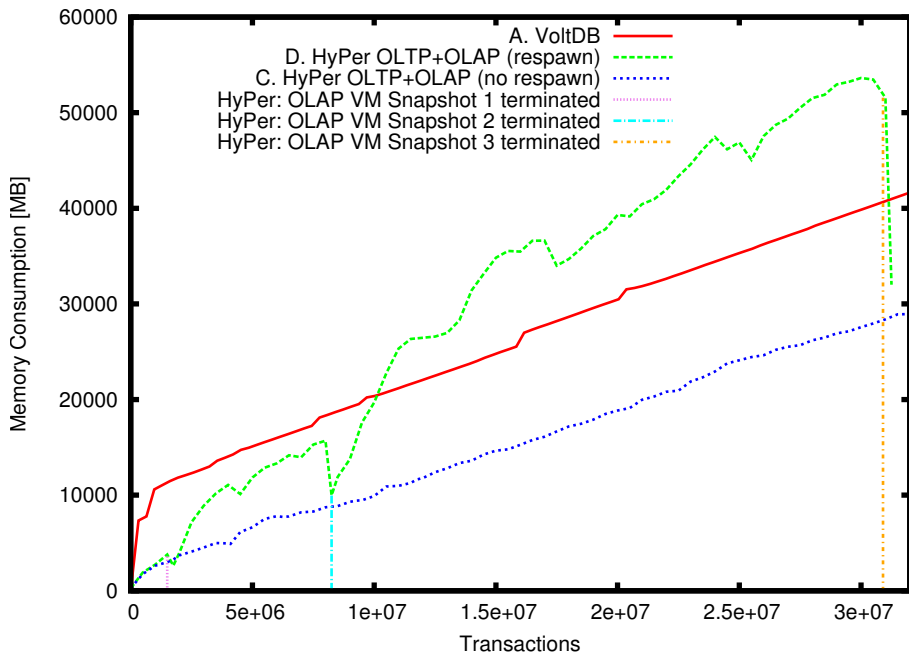


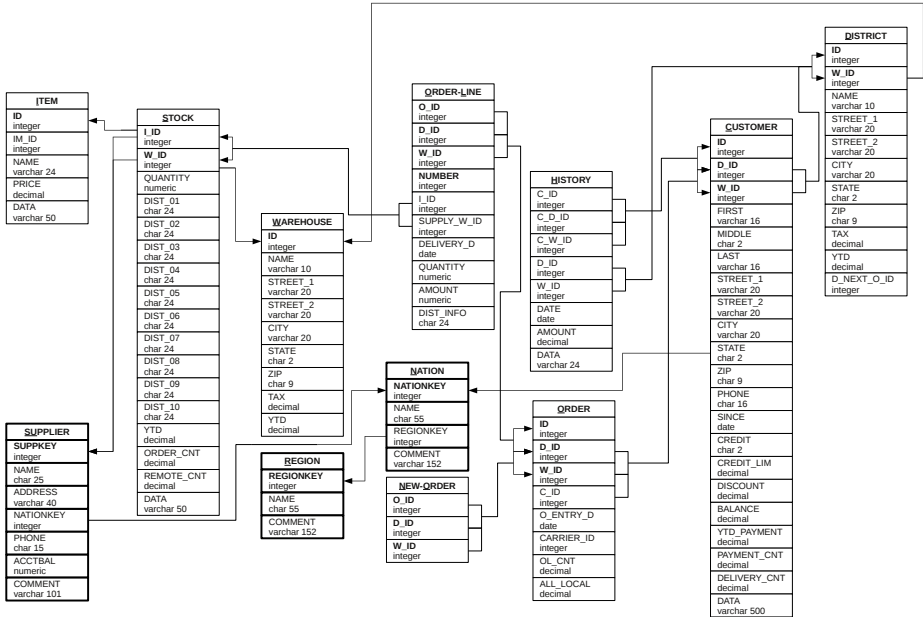
Figure 9: Memory consumption (after load) of VoltDB and HyPer

6 Summary

We presented TPC-CH, a benchmark for hybrid OLTP&OLAP database systems. TPC-CH is based on the standardized TPC-C and TPC-H benchmarks. It is not only suited for hybrid DBMS, but also allows to compare them with systems specialized on either of the two workloads as well as traditional, universal systems. We substantiate this claim with results of representatives of each type of database system.

Acknowledgment We acknowledge the fruitful discussions about this benchmark during the Dagstuhl Workshop on “Robust Query Processing” (September 2010). Stefan Krompaß helped to implement the System X benchmark.

A Schema



B Queries

All dates, strings and ranges in the queries are examples only. Due to space constraints, we had to relinquish pretty-printing, but include the queries formatted suitable for copy&paste.

Q1: Generate orderline overview

```

SELECT ol_number, SUM(ol_quantity) AS sum_qty, SUM(ol_amount) AS
sum_amount, AVG(ol_quantity) AS avg_qty, AVG(ol_amount) AS
avg_amount, COUNT(*) AS count_order FROM orderline WHERE
ol_delivery_d > '2007-01-02 00:00:00.000000' GROUP BY ol_number
ORDER BY ol_number
  
```

Q2: Most important supplier/item-combinations (those that have the lowest stock level for certain parts in a certain region)

```

SELECT su_supkey, su_name, n_name, i_id, i_name, su_address, su_phone,
su_comment FROM item, supplier, stock, nation, region, (SELECT s_i_id
AS m_i_id, MIN(s_quantity) AS m_s_quantity FROM stock, supplier,
nation, region WHERE mod((s_w_id*s_i_id), 10000)=su_supkey AND
su_nationkey=n_nationkey AND n_regionkey=r_regionkey AND r_name
LIKE 'Europ%' GROUP BY s_i_id) m WHERE i_id=s_i_id AND mod((s_w_id
  
```

```

* s_i_id),10000)=su_supkey AND su_nationkey=n_nationkey AND
n_regionkey=r_regionkey AND i_data LIKE '%b' AND r_name LIKE '
Europ%' AND i_id=m_i_id AND s_quantity=m_s_quantity ORDER BY
n_name,su_name,i_id

```

Q3: Unshipped orders with highest value for customers within a certain state

```

SELECT ol_o_id,ol_w_id,ol_d_id,SUM(ol_amount) AS revenue,o_entry_d
FROM customer,neworder,"order",orderline WHERE c_state LIKE 'A%'
AND c_id=o_c_id AND c_w_id=o_w_id AND c_d_id=o_d_id AND no_w_id=
o_w_id AND no_d_id=o_d_id AND no_o_id=o_id AND ol_w_id=o_w_id AND
ol_d_id=o_d_id AND ol_o_id=o_id AND o_entry_d>'2007-01-02
00:00:00.000000' GROUP BY ol_o_id,ol_w_id,ol_d_id,o_entry_d ORDER
BY revenue DESC,o_entry_d

```

Q4: Orders that were partially shipped late

```

SELECT o_ol_cnt,COUNT(*) AS order_count FROM "order" WHERE
o_entry_d>='2007-01-02 00:00:00.000000' AND o_entry_d<'2012-01-02
00:00:00.000000' AND EXISTS (SELECT * FROM orderline WHERE o_id=
ol_o_id AND o_w_id=ol_w_id AND o_d_id=ol_d_id AND ol_delivery_d>=
o_entry_d) GROUP BY o_ol_cnt ORDER BY o_ol_cnt

```

Q5: Revenue volume achieved through local suppliers

```

SELECT n_name,SUM(ol_amount) AS revenue FROM customer,"order",
orderline,stock,supplier,nation,region WHERE c_id=o_c_id AND
c_w_id=o_w_id AND c_d_id=o_d_id AND ol_o_id=o_id AND ol_w_id=
o_w_id AND ol_d_id=o_d_id AND ol_w_id=s_w_id AND ol_i_id=s_i_id
AND mod((s_w_id * s_i_id),10000)=su_supkey AND ascii(SUBSTRING(
c_state,1,1))=su_nationkey AND su_nationkey=n_nationkey AND
n_regionkey=r_regionkey AND r_name='Europe
' AND o_entry_d>='
2007-01-02 00:00:00.000000' GROUP BY n_name ORDER BY revenue DESC

```

Q6: Revenue generated by orderlines of a certain quantity

```

SELECT SUM(ol_amount) AS revenue FROM orderline WHERE
ol_delivery_d>='1999-01-01 00:00:00.000000' AND ol_delivery_d<'
2020-01-01 00:00:00.000000' AND ol_quantity BETWEEN 1 AND 100000

```

Q7: Bi-directional trade volume between two nations

```

SELECT su_nationkey AS supp_nation,cust_nation,o_year,SUM(
ol_amount) AS revenue FROM supplier,stock,orderline,(SELECT o_w_id
,o_d_id,o_id,o_c_id,EXTRACT(YEAR FROM o_entry_d) AS o_year FROM "
order") o,(SELECT c_id,c_w_id,c_d_id,c_state,SUBSTRING(c_state
,1,1) AS cust_nation FROM customer) c,nation nl,(SELECT
n_nationkey,n_name,code(n_nationkey) AS n_nationkeyasc FROM nation
) n2 WHERE ol_supplier_w_id=s_w_id AND ol_i_id=s_i_id AND mod((
s_w_id * s_i_id),10000)=su_supkey AND ol_w_id=o_w_id AND ol_d_id=
o_d_id AND ol_o_id=o_id AND c_id=o_c_id AND c_w_id=o_w_id AND

```

```

c_d_id=o_d_id AND su_nationkey=n1.n_nationkey AND cust_nation=n2.
n_nationkeyasc AND ((n1.n_name='Germany' AND n2.
n_name='Cambodia') OR (n1.n_name='Cambodia
' AND n2.n_name='Germany')) AND
ol_delivery_d BETWEEN TIMESTAMP '2000-01-01' AND TIMESTAMP '
2099-01-01' GROUP BY su_nationkey,cust_nation,o_year ORDER BY
su_nationkey,cust_nation,o_year

```

Q8: Market share of a given nation for customers of a given region for a given part type

```

SELECT EXTRACT (YEAR FROM o_entry_d) AS l_year,SUM(CASE WHEN n2.
n_name='Germany' THEN ol_amount ELSE 0 END)/SUM(
ol_amount) AS mkt_share FROM item,supplier,stock,orderline,"order"
,customer,nation n1,nation n2,region WHERE i_id=s_i_id AND ol_i_id
=s_i_id AND ol_supplier_w_id=s_w_id AND mod((s_w_id * s_i_id)
,10000)=su_supkey AND ol_w_id=o_w_id AND ol_d_id=o_d_id AND
ol_o_id=o_id AND c_id=o_c_id AND c_w_id=o_w_id AND c_d_id=o_d_id
AND n1.n_nationkey=ascii(SUBSTRING(c_state,1,1)) AND n1.
n_regionkey=r_regionkey AND ol_i_id<1000 AND r_name='Europe
' AND su_nationkey
=n2.n_nationkey AND o_entry_d BETWEEN '2007-01-02 00:00:00.000000'
AND '2012-01-02 00:00:00.000000' AND i_data LIKE '%b' AND i_id=
ol_i_id GROUP BY l_year ORDER BY l_year

```

Q9: Profit made on a given line of parts,broken out by supplier nation and year

```

SELECT n_name,EXTRACT (YEAR FROM o_entry_d) AS l_year,SUM(ol_amount
) AS sum_profit FROM item,stock,supplier,orderline,"order",nation
WHERE ol_i_id=s_i_id AND ol_supplier_w_id=s_w_id AND mod((s_w_id *
s_i_id),10000)=su_supkey AND ol_w_id=o_w_id AND ol_d_id=o_d_id
AND ol_o_id=o_id AND ol_i_id=i_id AND su_nationkey=n_nationkey AND
i_data LIKE '%BB' GROUP BY n_name,l_year ORDER BY n_name,l_year
DESC

```

Q10: Customers who received their ordered products late

```

SELECT c_id,c_last,SUM(ol_amount) AS revenue,c_city,c_phone,n_name
FROM customer,"order",orderline,nation WHERE c_id=o_c_id AND
c_w_id=o_w_id AND c_d_id=o_d_id AND n_nationkey=ascii(SUBSTRING(
c_state,1,1)) AND ol_w_id=o_w_id AND ol_d_id=o_d_id AND ol_o_id=
o_id AND o_entry_d>='2007-01-02 00:00:00.000000' AND o_entry_d<=
ol_delivery_d GROUP BY c_id,c_last,c_city,c_phone,n_name ORDER BY
revenue DESC

```

Q11: Most important (high order count compared to the sum of all ordercounts) parts supplied by suppliers of a particular nation

```

SELECT s_i_id,SUM(s_quantity) AS ordercount FROM stock,supplier,
nation WHERE mod((s_w_id * s_i_id),10000)=su_supkey AND
su_nationkey=n_nationkey AND n_name='Germany'
GROUP BY s_i_id HAVING SUM(s_quantity)>(SELECT SUM(s_quantity) *
.0001 FROM stock,supplier,nation WHERE mod((s_w_id * s_i_id)

```

```
,10000)=su_supkey AND su_nationkey=n_nationkey AND n_name='
Germany ' ) ORDER BY ordercount DESC
```

Q12: Determine whether selecting less expensive modes of shipping is negatively affecting the critical-priority orders by causing more parts to be received late by customers

```
SELECT o_ol_cnt,SUM(CASE WHEN o_carrier_id=1 OR o_carrier_id=2
THEN 1 ELSE 0 END) AS high_line_count,SUM(CASE WHEN o_carrier_id
<>1 AND o_carrier_id<>2 THEN 1 ELSE 0 END) AS low_line_count FROM
"order",orderline WHERE ol_w_id=o_w_id AND ol_d_id=o_d_id AND
ol_o_id=o_id AND o_entry_d<= ol_delivery_d AND ol_delivery_d<'
2020-01-01 00:00:00.000000' GROUP BY o_ol_cnt ORDER BY o_ol_cnt
```

Q13: Relationships between customers and the size of their orders

```
SELECT c_count,COUNT(*) AS custdist FROM (SELECT c_id,COUNT(o_id)
FROM customer LEFT OUTER JOIN "order" ON (c_w_id=o_w_id AND c_d_id
=o_d_id AND c_id=o_c_id AND o_carrier_id>8) GROUP BY c_id) AS
c_orders (c_id,c_count) GROUP BY c_count ORDER BY custdist DESC,
c_count DESC
```

Q14: Market response to a promotion campaign

```
SELECT 100.00 * SUM(CASE WHEN i_data LIKE 'PR%' THEN ol_amount
ELSE 0 END) / 1+SUM(ol_amount) AS promo_revenue FROM orderline,
item WHERE ol_i_id=i_id AND ol_delivery_d>='2007-01-02
00:00:00.000000' AND ol_delivery_d<'2020-01-02 00:00:00.000000'
```

Q15: Determines the top supplier

```
with revenue (supplier_no,total_revenue) AS (SELECT supplier_no,
SUM(ol_amount) FROM orderline, (SELECT s_w_id,s_i_id,mod((s_w_id *
s_i_id),10000) AS supplier_no FROM stock) s WHERE ol_i_id=s_i_id
AND ol_supplier_w_id=s_w_id AND ol_delivery_d>='2010-05-23
12:00:00' GROUP BY supplier_no) SELECT su_supkey,su_name,
su_address,su_phone,total_revenue FROM supplier,revenue WHERE
su_supkey=supplier_no AND total_revenue=(SELECT MAX(total_revenue
) FROM revenue) ORDER BY su_supkey
```

Q16: Number of suppliers that can supply parts with given attributes

```
SELECT i_name,brand,i_price,COUNT(DISTINCT (mod((s_w_id * s_i_id)
,10000))) AS supplier_cnt FROM stock, (SELECT i_id,i_data,i_name,
SUBSTRING(i_data,1,3) AS brand,i_price FROM item) i WHERE i_id=
s_i_id AND i_data NOT LIKE 'zz%' AND (mod((s_w_id * s_i_id),10000)
) NOT IN (SELECT su_supkey FROM supplier WHERE su_comment LIKE '%
bad%') GROUP BY i_name,brand,i_price ORDER BY supplier_cnt DESC
```

Q17: Average yearly revenue that would be lost if orders were no longer filled for small quantities of certain parts

```
SELECT SUM(ol_amount) / 2.0 AS avg_yearly FROM orderline,item
WHERE ol_i_id=i_id AND i_data LIKE '%b' AND ol_quantity<(SELECT
0.2 * AVG(ol_quantity) FROM orderline WHERE ol_i_id=i_id)
```

Q18: Rank customers based on their placement of a large quantity order

```
SELECT c_last,c_id,o_id,o_entry_d,o_ol_cnt,SUM(ol_amount) FROM
customer,"order",orderline WHERE c_id=o_c_id AND c_w_id=o_w_id AND
c_d_id=o_d_id AND ol_w_id=o_w_id AND ol_d_id=o_d_id AND ol_o_id=
o_id GROUP BY o_id,o_w_id,o_d_id,c_id,c_last,o_entry_d,o_ol_cnt
HAVING SUM(ol_amount)>200 ORDER BY SUM(ol_amount) DESC,o_entry_d
```

Q19: Machine generated data mining (revenue report for disjunctive predicate)

```
SELECT SUM(ol_amount) AS revenue FROM orderline,item WHERE (
ol_i_id=i_id AND i_data LIKE '%a' AND ol_quantity>=1 AND
ol_quantity<= 10 AND i_price BETWEEN 1 AND 400000 AND ol_w_id IN
(1,2,3)) OR (ol_i_id=i_id AND i_data LIKE '%b' AND ol_quantity>=1
AND ol_quantity<= 10 AND i_price BETWEEN 1 AND 400000 AND ol_w_id
IN (1,2,4)) OR (ol_i_id=i_id AND i_data LIKE '%c' AND ol_quantity
>=1 AND ol_quantity<= 10 AND i_price BETWEEN 1 AND 400000 AND
ol_w_id IN (1,5,3))
```

Q20: Suppliers in a particular nation having selected parts that may be candidates for a promotional offer

```
SELECT su_name,su_address FROM supplier,nation WHERE su_suppkey IN
(SELECT mod(s_i_id * s_w_id,10000) FROM stock,orderline WHERE
s_i_id IN (SELECT i_id FROM item WHERE i_data LIKE 'co%') AND
ol_i_id=s_i_id AND ol_delivery_d>'2010-05-23 12:00:00' GROUP BY
s_i_id,s_w_id,s_quantity HAVING 2*s_quantity>SUM(ol_quantity)) AND
su_nationkey=n_nationkey AND n_name='Germany'
ORDER BY su_name
```

Q21: Suppliers who were not able to ship required parts in a timely manner

```
Q21: SELECT su_name,COUNT(*) AS numwait FROM supplier,orderline l1
,"order",stock,nation WHERE ol_o_id=o_id AND ol_w_id=o_w_id AND
ol_d_id=o_d_id AND ol_w_id=s_w_id AND ol_i_id=s_i_id AND mod((
s_w_id * s_i_id),10000)=su_suppkey AND l1.ol_delivery_d>o_entry_d
AND NOT EXISTS (SELECT * FROM orderline l2 WHERE l2.ol_o_id=l1.
ol_o_id AND l2.ol_w_id=l1.ol_w_id AND l2.ol_d_id=l1.ol_d_id AND l2
.ol_delivery_d>l1.ol_delivery_d) AND su_nationkey=n_nationkey AND
n_name='Germany' GROUP BY su_name ORDER BY
numwait DESC,su_name
```

Q22: Geographies with customers who may be likely to make a purchase

```
SELECT country,COUNT(*) AS numcust,SUM(c.c_balance) AS totacctbal
FROM (SELECT c_phone,c_balance,c_id,c_w_id,c_d_id,c_balance,
SUBSTRING(c_state,1,1) AS country FROM customer) c WHERE SUBSTRING
(c_phone,1,1) IN ('1','2','3','4','5','6','7') AND c.c_balance>(
SELECT AVG(c2.c_balance) FROM customer c2 WHERE c2.c_balance>0.00
AND SUBSTRING(c2.c_phone,1,1) IN ('1','2','3','4','5','6','7'))
AND NOT EXISTS (SELECT * FROM "order" WHERE o_c_id=c_id AND o_w_id
=c_w_id AND o_d_id=c_d_id) GROUP BY country ORDER BY country
```

C Relations

NATION					
NATION-KEY	NAME	REGION-KEY	NATION-KEY	NAME	REGION-KEY
48	Australia	4	86	Finnland	5
49	Belgium	5	87	Ghana	1
50	Cameroon	1	88	Haiti	2
51	Denmark	5	89	India	3
52	Ecuador	2	90	Jamaica	4
53	France	5	97	Kazakhstan	3
54	Germany	5	98	Luxembourg	5
55	Hungary	5	99	Morocco	1
56	Italy	5	100	Norway	5
57	Japan	3	101	Poland	5
65	Kenya	1	102	Peru	2
66	Lithuania	5	103	Nicaragua	2
67	Mexico	2	104	Romania	5
68	Netherlands	5	105	South Africa	1
69	Oman	1	106	Thailand	3
70	Portugal	5	107	United Kingdom	5
71	Qatar	1	108	Venezuela	2
72	Rwanda	1	109	Liechtenstein	5
73	Serbia	5	110	Austria	5
74	Togo	1	111	Laos	3
75	United States	2	112	Zambia	1
76	Vietnam	3	113	Switzerland	5
77	Singapore	3	114	China	3
78	Cambodia	3	115	Papua New Guinea	4
79	Yemen	1	116	East Timor	4
80	Zimbabwe	1	117	Bulgaria	5
81	Argentina	2	118	Brazil	2
82	Bolivia	2	119	Albania	5
83	Canada	2	120	Andorra	5
84	Dominican Republic	2	121	Belize	2
85	Egypt	1	122	Botswana	1

REGION		
REGIONKEY	NAME	...
1	Africa	
2	America	
3	Asia	
4	Australia	
5	Europe	

References

- [BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. *International Conference on Management of Data*, 2009.
- [BKS08] Anja Bog, Jens Krüger, and Jan Schaffner. A Composite Benchmark for Online Transaction Processing and Operational Reporting. *2008 IEEE Symposium on Advanced Management of Information for Globalized Enterprises (AMIGE)*, September 2008.
- [BMK09] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2), 2009.
- [CS04] Sang K Cha and Changbin Song. P * TIME : Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, 2004.
- [DHKK97] Jochen Doppelhammer, Thomas Höppler, Alfons Kemper, and Donald Kossmann. Database performance in the real world: TPC-D and SAP R/3. *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, 1997.
- [GUMG10] Georgios Giannikis, Philipp Unterbrunner, Jeremy Meyer, and G. Crescando. *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, 2010.
- [HAMS08] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, 2008.
- [KGT⁺10] Jens Krueger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Franz Faerber. Optimizing Write Performance for Read Optimized Databases. In *Database Systems for Advanced Applications*. Springer, 2010.
- [KKN⁺08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2), 2008.
- [KN11] Alfons Kemper and Thomas Neumann. HyPer – A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. *ICDE 2011: IEEE International Conference on Data Engineering*, 2011.
- [Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, 2009.
- [SBKZ08] Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. A Hybrid Row-Column OLTP Database Architecture for Operational Reporting. *Business Intelligence for the Real Time Enterprise (BIRTE 2008)*, 2008.
- [Tra10a] Transaction Processing Performance Council. TPC Benchmark C (Standard Specification), 2010.
- [Tra10b] Transaction Processing Performance Council. TPC Benchmark H (Standard Specification), 2010.
- [Tra10c] Transaction Processing Performance Council. TPC-C Results. http://www.tpc.org/downloaded_result_files/tpcc_results.txt, 2010.
- [Tra10d] Transaction Processing Performance Council. TPC-E Results. http://www.tpc.org/downloaded_result_files/tpce_results.txt, 2010.
- [Vola] VoltDB TPC-C-like Benchmark Comparison-Benchmark Description. <http://community.voltdb.com/node/134>.
- [Volb] VoltDB Inc. VoltDB: Product Overview.

Simulating Multi-Tenant OLAP Database Clusters

Jan Schaffner¹, Benjamin Eckart¹, Christian Schwarz¹, Jan Brunnert¹, Dean Jacobs²,
Alexander Zeier¹, and Hasso Plattner¹

¹Hasso Plattner Institute, University of Potsdam, August-Bebel-Str. 88, 14482 Potsdam,
Germany, Email: {firstname.lastname}@hpi.uni-potsdam.de

²SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany, Email:
{firstname.lastname}@sap.com

Abstract: Simulation of parallel database machines was used in many database research projects during the 1990ies. One of the main reasons why simulation approaches were popular in that time was the fact that clusters with hundreds of nodes were not as readily available for experimentation as it is the case today. At the same time, the simulation models underlying these systems were fairly complex since they needed to capture both queuing processes in hardware (e.g. CPU contention or disk I/O) and software (e.g. processing distributed joins). Today's trend towards more specialized database architectures removes large parts of this complexity from the modeling task. As the main contribution of this paper, we discuss how we developed a simple simulation model of such a specialized system: a multi-tenant OLAP cluster based on an in-memory column database. The original infrastructure and testbed was built using SAP TREX, an in-memory column database part of SAP's business warehouse accelerator, which we ported to run on the Amazon EC2 cloud. Although we employ a simple queuing model, we achieve good accuracy. Similar to some of the parallel systems of the 1990ies, we are interested in studying different replication and high-availability strategies with the help of simulation. In particular, we study the effects of mirrored vs. interleaved replication on throughput and load distribution in our cluster of multi-tenant databases. We show that the better load distribution inherent to the interleaved replication strategy is exhibited both on EC2 and in our simulation environment.

1 Introduction

Implementing distributed systems and conducting experiments on top of them is usually both difficult and a lot of work is required to “get things right”. When conducting research on a distributed system, such as for e.g. a multi-node database cluster, the turnaround time for changing an aspect of the system's design from implementation to testing is thus often high. At the same time, research on distributed systems is often experimental, i.e. the cycle of implementing and validating ideas on different system designs is repeated fairly often.

The simulation of software systems can serve as one possible tool to shortcut the evaluation of system designs, although it cannot replace building (and experimenting with) actual systems. Especially in the light of new hardware becoming available and being deployed

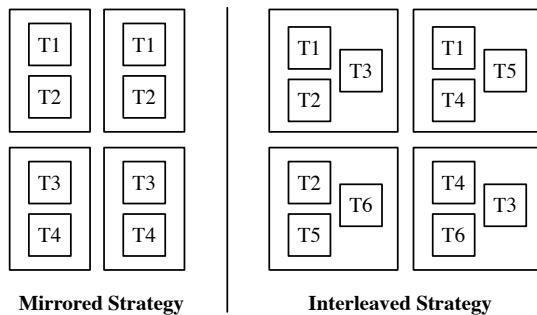


Figure 1: Example Layouts of Tenant Data

at cloud infrastructure providers simulation allows the prediction of cluster behavior based on predicted performance increases on a cloud platform. In fact, simulation models were a prominent means to evaluate parallel database systems such as Bubba [BAC⁺90] and Gamma [DGS⁺90] in the late 1980ies and 1990ies. These simulation models underlying these systems are fairly complex, since they capture the most important components of computer systems and their inter-dependencies, from the CPU and the main-memory sub-system to disk and network I/O, not to forget the multitude of software components involved in database query processing. The recent trend in database research towards specialized systems with simplified architectures [Sto08] does, however, also simplify the creation of simulation models.

In this paper, we describe our experience with building a simulation model for a multi-tenant OLAP cluster based on TREX, SAP’s in-memory column database [Pla09, SBKZ08, JLF10]. TREX was designed to support interactive business intelligence applications that require a) sub-second response times for ad-hoc queries to facilitate exploratory analysis and b) incremental insertions of new data to provide real-time visibility into operational processes. In previous work, we ported TREX to run in the Amazon EC2 cloud [Ama] and built a clustering framework round TREX, called Rock, that supports multi-tenancy, replication, and high availability.

In-memory databases perform disk I/O only during write transactions to ensure durability. Our data warehousing workload is however read-mostly in the sense that writes occur only during ETL periods and have batch character. Also, column-databases are known to be CPU-bound for scan-intensive workloads (such as for e.g. data warehousing) [SAB⁺05]. All this allows us to build a much simpler simulation model which is yet accurate in comparison to execution traces of the real system.

Similar to some of the parallel systems of the 1990ies, we are interested in studying different replication and high-availability strategies with the help of simulation. This paper experimentally compares two data placement strategies for Analytic Databases in a Cloud Computing environment, mirroring and interleaving. Example layouts from these strategies are shown in Figure 1, where the large boxes represent databases and the small boxes within them represent data for individual tenants.

When using the mirrored strategy, two copies of each database are maintained, both of which contain the same group of tenants. To ensure acceptable response times during recovery periods, each server must have sufficient capacity to handle the entire workload on its own, thus the system must be 100% over-provisioned [MS03]. This strategy is used by many on-demand services today.

Two copies of each tenant’s data are maintained, when using the interleaving strategy. The data is distributed across the cluster so as to minimize the number of pairs of tenants that occur together on more than one server. This strategy reduces the amount of over-provisioning that is required to handle failures and load surges because the excess work is distributed across many other servers.

We show that, without failures or variations in the request rate, the interleaved strategy achieves higher throughput than the mirrored strategy. For the moderately-sized tenants used in our experiments in the real system, the improvement is 7%. This improvement occurs because the interleaved strategy smoothes out statistical variations in the workload that depend on which queries are submitted to which servers. We wanted to make sure that this effect is a result of the chosen placement strategy and not a random effect coming from random variations in capacity of Amazon EC2 VMs. We therefore parameterized our simulator with a similar setup and were able to produce a similar result. We also evaluate the impact of server crashes for both mirrored and interleaving on the real cluster and using simulation.

This paper is organized as follows: Section 2 describes the Rock clustering infrastructure. Section 3 introduces the benchmark which was used for all experiments in this paper. Section 4 we analyze the requirements and discuss our implementation of our discrete event simulator based on the Rock clustering infrastructure and the benchmark. Section 5 discusses our data placement experiments both in the real system and the simulator. Section 6 discusses related work. Section 7 concludes the paper.

2 The Rock Framework

The Rock clustering framework runs in front of a collection of TREX servers and provides multi-tenancy, replication of tenant data, and fault tolerance. Figure 2 illustrates the architecture of the Rock framework. Read requests are submitted to the cluster by the analytics application. Write requests are submitted by the batch importers, which periodically pull incremental updates of the data from transactional source systems. The Rock framework itself consists of three types of processes: the *cluster leader*, *routers*, and *instance managers*. Each instance manager is paired one-to-one with a TREX server to which it forwards requests.

The cluster leader exists only once in the landscape and assigns tenant data to instance managers. The cluster leader as well as the batch importer are assumed to be highly available by replicating state using the Paxos[Lam98] algorithm, which would provide fail-safe distributed state for these critical components. The actual implementation is considered future work at this point. Each copy of a tenant’s data is assigned to one instance manager

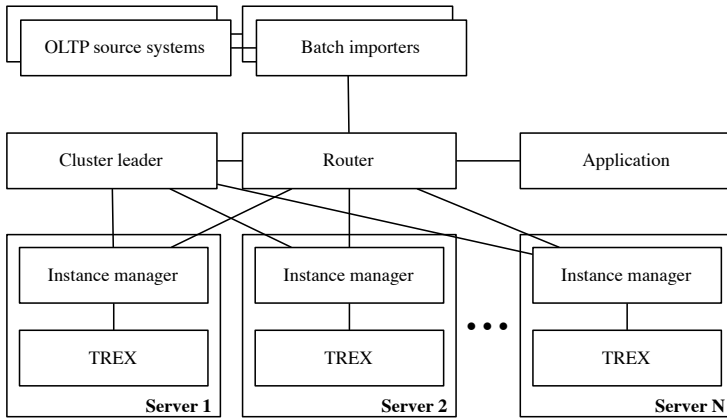


Figure 2: The Rock Analytic Cluster Architecture

and each instance manager is responsible for the data from multiple tenants. The cluster leader maintains the assignment information in a *cluster map*, which it propagates to the routers and instance managers so all components share a consistent view of the landscape. The cluster leader tracks changes to the state of the cluster based on information it collects from the Amazon EC2 API such as IP addresses, instance states, and geographic location. The cluster leader is not directly involved in request processing.

The routers accept requests from outside the cluster and forward them to the appropriate instance managers. Routing is based on the tenant who issued the query and the chosen load balancing strategy. Our current implementation supports round-robin, random, and server-load-based load balancing. The experiments in this paper use the latter algorithm. Load is taken to be the CPU idle time of the TREX server averaged over a 10 second window. The small window size is crucial for the router’s ability to re-direct queries to the least utilized replica during a load burst. Load information is piggy-backed onto query results as they are returned to the router.

Rock offers master/master replication [GHOS96]: a router may forward a write request to any one of the instance managers for a tenant, which then propagates the write to the other instance managers for that tenant. We assume there is a single batch importer per tenant and that writes are sequentially numbered, thus master/master replication is straightforward to implement without introducing inconsistencies in the data. Read consistency is required to support multi-query drill down into a data set, and TREX implements it using multi-version concurrency control (MVCC) based on snapshot isolation [BBG⁺95].

According to [JA07], multi tenancy can be realized in the database by adopting a *shared-machine*, *shared-process*, or *shared-table* approach. The shared-table approach, where each table has a `tenant_id` column, can be made efficient if accesses are index-based. However analytic queries on column databases generally entail table scans, and scan times are proportional to the number of rows in the table. Rock therefore uses the shared-process approach and gives each tenant their own private tables.

3 Experiments on the Amazon EC2 Cloud

The experiments in this paper are based on a modified version of the Star Schema Benchmark (SSB) [OOC07], which is an adaptation of TPC-H [TPC].

To produce data for our experiments, we used the data generator of SSB, which is based on the TPC-H data generator. As stated in Section 2, we give each tenant their own private tables, thus there is one instance of the SSB data model per tenant. In the experiments presented in this paper, all tenants have the same size, i.e. 6,000,000 rows in the fact table. As a point of comparison, a Fortune 500 consumer products and goods enterprise with a wholesale infrastructure produces about 120 million sales order line items per year, which is only a factor of 20 greater than the tenant size chosen for this paper. Using TREX’s standard dictionary compression, the fully-compressed data set consumes 204 MB in main memory.

While TPC-H has 22 independent data warehousing queries, SSB has four *query flights* with three to four queries each. A query flight models a drill-down, i.e. all queries compute the same aggregate measure but use different filter criteria on the dimensions. This structure models the exploratory interactions of users with business intelligence applications. We modified SSB so all queries within a flight are performed against the same TREX transaction ID to ensure that a consistent snapshot is used.

In our benchmark, each tenant has multiple concurrent *users* that submit requests to the system. Each user cycles through the query flights, stepping through the queries in each flight. After receiving a response to a query, a user waits for a fixed think time before submitting the next query. To prevent caravanning, each user is offset in the cycle by a random amount.

The number of users for a given tenant is taken to be the size of that tenant multiplied by a scale factor. Our experiments vary this scale factor to set the overall rate of requests to the system. In reporting results, we give the maximum number of simultaneous users rather than the throughput, since users are the basis of pricing and revenue in the Software as a Service setting. TPC-DS also models concurrent users and think times [PSKL02]. Following [SPvSA07], which studies web applications, we draw user think times from a negative exponential distribution with a mean of five seconds.

A benchmark run is evaluated as follows. The first ten minutes are cut off to ensure that the system is warmed up. The next ten minutes after the warmup are called the benchmark period. All queries submitted after the benchmark period are cut off as well. A run of the benchmark is considered to be successful only if, during the benchmark period, the response times at the 99-th percentile of the distribution are within one second. Response times are measured at the router. Sub-second response times are essential to encourage interactive exploration of a dataset and, in any case, have become the norm for web applications regardless of how much work they perform. The focus on performance at the 99-th percentile is also common; see [DHJ⁺07] for example.

The results presented in this paper are highly dependent on specific configuration choices described in this section. Nevertheless we believe these results are applicable in most practical situations. Our tenants are relatively large by SaaS standards and, for smaller tenants,

interleaving would distribute excess work more evenly across the cluster. Five second think times are perhaps too short for more complex applications, but the system behaves linearly in this respect: doubling the think time would double the maximum number of simultaneous users.

All experiments are run on *large memory* instances on Amazon EC2, which have 2 virtual compute units (i.e. CPU cores) with 7.5 GB RAM each. For disk storage, we use Amazon EBS volumes, which offers highly-available persistent storage for EC2 instances. The disks have a performance impact only on recovery times. An EBS volume can be attached to only one EC2 instance at a time.

4 Simulation Model

For the simulation, we need to model the real system and benchmark, which have been described in Sections 2 and 3. In this section, we analyze the requirements and discuss our implementation of a discrete event simulator.

4.1 Problem Statement

Given a special-purpose clustering framework (Rock) and a commercial in-memory database system (TREX), we can assess the viability of using simulation techniques to estimate the performance characteristics exhibited by such a system. The goal is to accurately model the most relevant environmental parameters as well as the different load balancing, data placement and high availability techniques employed in the real cluster system.

The simulation should provide results that allow a relevant assessment of various strategies in the context of a cluster setup. The accuracy of the simulation results shall be validated against the empirical results for a static cluster configuration. The simulation does not take into account message passing latency between system components or network bandwidth, but focuses on the kernel execution time of the in-memory column database, which is composed of CPU execution time and time waiting for the operating system to schedule a CPU for the execution thread.

We will begin with discussing the fundamentals of the simulation model, such as the modeling of the query processing components and the user load model as well as describing the implementation of the simulator. The simulation results will be presented in the following section.

4.2 Simulation Model of the In-Memory Database Cluster

Discrete Event Simulation using a process-oriented paradigm allows an integrated simulation of the most important components and processes in the cluster. In a process-oriented

simulation model, different active components are modularized in processes. The execution of parallel processes is serialized by explicit wait statements that allow simulation time to skip ahead to the next occurring event. This approach is more modular and CPU efficient than the activity oriented paradigm and, therefore, allows simulation of user activities using a more fine-grained queuing model of the involved components and their users.

The simulation model consists of *resources* and *processes*. In the case of the simulated Rock cluster, the resources are compute *Nodes* (virtual machine instances running instance manager/TREX pairs). Nodes have an immutable number of processors and amount of main memory. It is assumed that these virtual machines are used for serving database requests exclusively. Queues are established when simulation processes need to wait for a shared resource.

Processes are actors within the simulator. For example, the activity of a single user, of which there are multiple for each tenant data-set, is modeled as a *User* process. Multiple processes of the User type are active in parallel. Users create *Query* processes that simulate the execution of queries on the limited Node resources.

To simulate a behaviour of a system, we have to understand and model the behaviour of a system. One common approach to model a system is *Queueing Network Modeling*. According to Lazowska et al., *Queueing Network Modeling* is a an approach in which a computer system is represented as a network of queues which is evaluated analytically [LZGS84]. A network of queues consists of several *service centers* which are system resources and are used by *customers* that are the users of the system. If customers arrive at a higher rate than the service center can handle, customers are queued. The time which is necessary for a transaction to be finished is, then, not only the time the service center requires, but also the waiting time in the queue.

If more queries arrive than can be executed by all query threads, subsequent queries will be queued. In a queueing network model, each query thread is represented by a service center. Each query thread uses one of the two CPUs which itself are represented by two further service centers. The threads are sharing the processing unit resources using a time slice model.

4.3 Modeling Query Processing Components

The goal of the simulation is to model a cluster of in-memory columnar database instances. The cluster's response time profile has been studied empirically using the SSB benchmark, which yields as raw data the query processing times for individual requests. At the core of the discrete event simulation is the statistical model of the kernel execution times, or service-center processing times, based on query type. For the purpose of establishing the internal processing times, we have analyzed a long-running benchmark on the experimental framework with only a single user in order to establish a baseline without queuing interference. Based on this data we determined which statistical distribution best matches the real distribution. In general, one often uses exponential distributions for "neutral"

simulation-to-simulation comparisons of scenarios, because the exponential distribution has favorable properties in regards to calculations. However, for modeling our Rock cluster infrastructure, it turns out that the gamma distribution is the best choice.

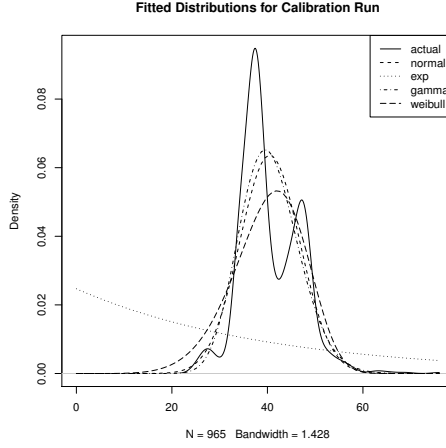


Figure 3: Distribution Fitting for Query 2.2

As can be seen for the example of SSB Query 2.2 in Figure 3, the distribution of query times from the single-user baseline run shows that the real distribution resembles a head-and-shoulders pattern, but has a strong peak around the mean processing time. We observe similar peaks for all other SSB query types as well. The diagram also shows how various distributions are fitted to the query.

We model the processing times of all queries fitting a gamma distribution to each query type using different parameters for shape k and scale Θ . Table 1 shows the corresponding parameters for each query. The following equation shows the gamma distribution:

$$f(x; k, \Theta) = x^{k-1} \frac{e^{-x/\Theta}}{\Theta^k \Gamma(k)}, \quad \text{for } x \geq 0 \text{ and } k, \Theta > 0 \quad (1)$$

One essential challenge is that the distribution of response times in the real cluster contains spikes, whereas statistical distributions typically look smooth. The gamma distribution is useful for our scenario, as it best resembles most of the queries shapes and allows us to smooth out the smaller spikes occurring in the real system. Still, the sample space remains usable because a greater variation is introduced around the mean due to the continuous random sampling in the simulator, which imitates the effect of the discrete hot-spots in the real system. The distribution drives a separate random number generator for each Node to generate internal kernel execution times for each query type. Because the sampled times from the calibration run are gross times that include networking and processing overheads, which are not part of the actual internal service center times, we establish an

internal speed-up factor¹, which is variably adapted for the baseline test, that shifts the distribution in favor of a faster internal execution, while preserving the system-inherent distribution characteristics. This approach has been superior to using a distribution based on the minimum response time, which did not accurately reflect the overheads resulting in occasional processing slowdowns in the real systems.

	Shape k	Scale Θ
Query 1.1	343.794	2.685
Query 1.2	18.452	0.685
Query 1.3	3.547	0.54
Query 2.1	188.744	2.257
Query 2.2	42.997	1.061
Query 2.3	15.319	0.564
Query 3.1	379.154	2.525
Query 3.2	96.046	1.595
Query 3.3	13.693	0.568
Query 3.4	12.536	0.529
Query 4.1	311.531	2.28
Query 4.2	70.306	0.636
Query 4.3	122.473	1.705

Table 1: Gamma Distribution Parameters for SSB Queries

4.4 Simulation Accuracy

In order to evaluate our simulation model’s accuracy we use a benchmark trace taken from experiments on a Rock cluster instance running on Amazon EC2 and compare the trace against the output of our simulation, which mimics the real system trace output. A trace contains query response times for all tenants’ users’ queries submitted during a 900 second test period, from which all data after a warmup period of 300 seconds is analyzed.

When comparing non-aggregated query execution times as shown in the Q-Q plot in Figure 4, one can see that the plot forms an almost identical line with the reference line, indicating that the individual query times generated by the simulation come from the same distribution as the execution times in the empirical system. This also validates our assumption that by closely modeling the underlying internal execution with a statistical distribution, reducing these times by measured overhead, and then adding queuing-theoretic waiting times, we can model the multi-tenant cluster with good accuracy.

The fact that the execution time plot is slightly above the reference line for faster queries shows that the real system has a larger fractional overhead for smaller queries than we are actually simulating. For a minority of slow queries, the simulator again returns too fast response times, indicating that these have a larger overhead on the real system, in spite of

¹The factor for the comparisons in this paper was 0.85

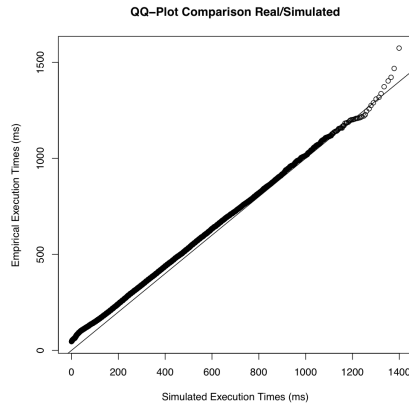


Figure 4: QQ-plot of simulated query times and actual cluster query times

our threading implementation, but this only applies to few queries. Nevertheless, the plots show that the simulator produces a very close match of the distribution indicated by the straight nature of the plot.

Therefore we can support our implementation goal to have a simulator that is accurate enough to enable the comparison of multiple scenarios with varying parameters and configurations to each other, while maintaining a close match to results obtained from real systems. This result is possible due to the very predictable performance characteristics of in-memory databases, due to the absence of complicated disk I/O scheduling.

Nevertheless, real systems have many influencing factors, which require re-calibration of the simulator after major changes in the underlying database software or virtualized PaaS environment. As a consequence, effects discovered in simulation still need to be backed by experiments on the real system.

	Simulation	Real execution
# Users	3500 users	4000 users
# Queries	397265	341560
Mean response time	305.9 ms	338.6 ms

Table 2: Maximum Number of Concurrent Users Before SLO Violation

We can see in the Table 2 that the number of queries in a given period of time is higher in the simulation than in the real test. The reason is that we are using an idealized model which is a strong simplification of the system. For example, some locking interdependencies that might cause queries to stall in the real system are not captured in the simulation model. We only model a single queue in front of the processors, which is not accurate since there is also queuing around network resources.

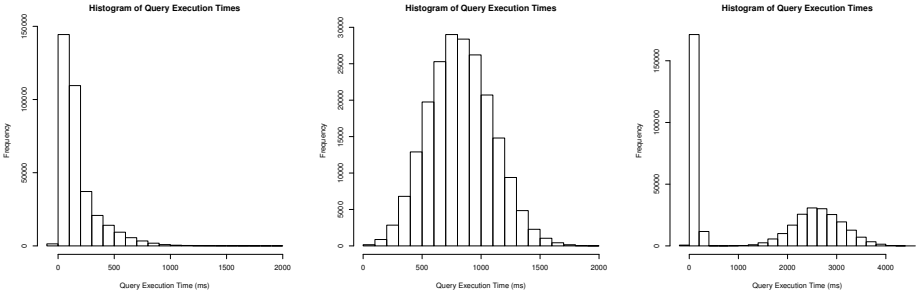
In summary, the response times of individual queries are accurately reproduced by the simulator, as shown in the Q-Q plot in Figure 4, while the total number of queries executed

within a benchmark run and the maximum number of users in the system before violating the response time goal are not perfectly aligned. However, the goal of our simulator is to test different cluster deployment options such as for e.g. mirrored vs. interleaved replica placement. We are interested in the relative performance difference of a choice of deployment options in the simulator. Our simulation model is suitable for this purpose, as we shall see in the next section.

5 Simulation Results

In this section, we analyze the simulation results and compare them with measurements conducted on our real system.

5.1 Distribution of Response Times Under Low and High Load



(a) Distribution in non-overload situation with threading (b) Distribution in overload situation without threading (c) Distribution in overload situation with threading

Figure 5: Query time histogram (frequency over response time in ms) for normal behavior (a), overload (b) and overload with threading (c)

The behavior of the simulated database cluster is greatly influenced by the cluster’s capacity in terms of available CPU resources. Without load spikes, the system shows a response time profile as shown in the histogram in Figure 5(a), looking much like an exponential distribution with many fast queries. The simulator can simulate time-slice scheduling, and we are currently using a time slice of 30 ms, modeled after the scheduling quantum of the adapted Xen environment of Amazon EC2. Of course, scheduling in real systems would also include the priority-based scheme of the operating system, with more complex interactions between processes, but in our experimental research system all processes are CPU-bound and not mixed with I/O bound tasks on the same kernel instance, allowing us to get our good matches between real and simulated benchmarks using the fixed time slice. Yet, when threading is enabled, the absolute number of queries processed during the overall simulation run is much higher (528413 vs 351422 queries), but the mean pro-

cessing time is also higher (180 ms vs. 125 ms mean). Threading allows faster queries to fast-track slow-running queries and therefore increases overall throughput at the expense of execution speed for slower queries.

When looking at a simulated run where we have increased the number of total users hitting the system at once beyond the capacity limit, the benefit of admitting more queries at once is clearly visible. While in Figure 5(b) the overload in the dedicated CPU system produces a load profile that is shaped like a normal distribution with a much higher mean. This is due to the fact that too many queries are queuing up in the system and are pushing all following queries up in their response time. The shared CPU run with 4 threads on 2 simulated CPUs in Figure 5(c) shows clearly that smaller queries are still being processed quickly, while only the slower queries suffer from the overload. Therefore, threading clearly reduces the visible latency for users with fast queries at the expense of those with slower queries.

Valuable insight can be gained from the behavior under overload conditions, when the resources required for all user requests exceed the available capacity. The way the system behaves in such overload conditions depends on system configuration parameters such as the maximum response time before a query is considered to have failed or how quickly additional resources can be acquired from the underlying cloud infrastructure. Another fundamental decision is whether to enable simulated time-slice multitasking in such a CPU-bound processing problem. The real system uses a maximum of four computing threads on a system with two virtual processors, therefore overcommitting the CPU resource while at the same time throttling the maximum number of parallel requests in the execution state. This two-times overcommitment has been shown to deliver the best results for the SSB workload and is explained by the fact that generated plan operations synchronize well before the activation of the next plan step.

5.2 Distribution of Response Times in the Presence of Failures

As stated in Section 2, Rock uses an active/active load balancing scheme in the presence of multiple replicas. If a server goes down, the workload which was handled by the crashed server is re-distributed to the servers holding the other copy of the tenants' data. The re-distribution of workload in the event of a server failure differs depending on how the tenant replicas are assigned to the servers in the cluster.

Using the off-the-shelf replication capabilities as offered by most modern databases would result on replicating the data on the granularity of a whole server. In doing so, all tenants appearing together on one server will also co-appear on a second server in the cluster. This technique is often referred to as *mirroring* (cf. Figure 1). The downside of mirroring is that in case of a failure all excess workload is re-directed to the other mirror server. In that case, the mirror server becomes a local hotspot in the cluster until the failed server is back online. A technique for avoiding such hotspots is to use *interleaving*, which was first introduced in Teradata [Ter85]. Interleaving entails performing replication on the granularity of the individual tenants rather than all tenants inside a database process. This allows for spreading out the excess workload in case of a server failure across multiple

machines in the cluster.

The following experiment in the real system demonstrates the impact of the chosen replica placement strategy on a cluster’s ability to serve queries without violating the SLO both during normal operations and failures: We set up a cluster with 100 tenants, where we put 10 tenants on each server. All tenants had exactly the same size (6 million rows in the fact) table and there were two copies per tenant, hence 20 servers in total. We assigned the tenant replicas to the server both using the mirrored strategy, where groups of 10 tenants were mirrored on one pair of servers each, and the interleaved strategy, where we manually laid out the tenants such that no two tenant replicas appear together on more than one server. Automatic generation of interleaved placements and incremental self-configuration of the cluster is ongoing research in our group and not in the scope of this paper, but discussed in our work on performance prediction[SEJ⁺ar]. We then ran both placement configurations under normal conditions and under failures. In the failure case, 1 out of the 20 TREX instances in the cluster was killed every 60 seconds. Given the average recovery time in our experiment, 1 out of 20 servers was thus unavailable for approximately 50% of the benchmark period in the failure case. Note that this is a very high failure rate which is unlikely to occur in practice.

Table 3 shows the results of the experiment on the EC2 cluster. Even under normal operating conditions, interleaving allows for 7% more throughput before the response time goal of one second in the 99th percentile is violated. The reason is that statistical variations occur when the number concurrently active users is high. These variations create short-lived load spikes, which the interleaved configuration spreads out better in the cluster than mirroring. As expected, the maximum throughput that the mirrored configuration can sustain in the failure case before an SLO violation occurs drops by almost 50% when compared to normal operations. Interleaving, in contrast, completely hides the failure from a throughput perspective. Notably, the interleaved configuration can even support 32 more users than the mirrored configuration without failures.

	Mirrored	Interleaved	Improvement
Normal operations	4218 users	4506 users	7%
Periodical failure	2265 users	4250 users	88%

Table 3: Maximum Number of Concurrent Users Before SLO Violation

On the real system in the Amazon EC2 cloud it could be shown that the layout, meaning how tenants are placed on the nodes in the cluster, had an impact on system performance, especially in the event of failures. We are interested in proving that this effect is a real property of the system, rather than a random effect which stems from external factors, such as for example non-uniformity in the capacity of the virtual machines procured by EC2. To do so, we enhance the queuing model of the simulator to model node crashes: The *Fault* is modeled as an optional process that can inject fault events into the query process based on its own failure distribution model. In our case we chose to inject failures at static times during the simulation, although an exponential distribution could also be used in repeated experiments to study the independence of failure behaviors and failure time.

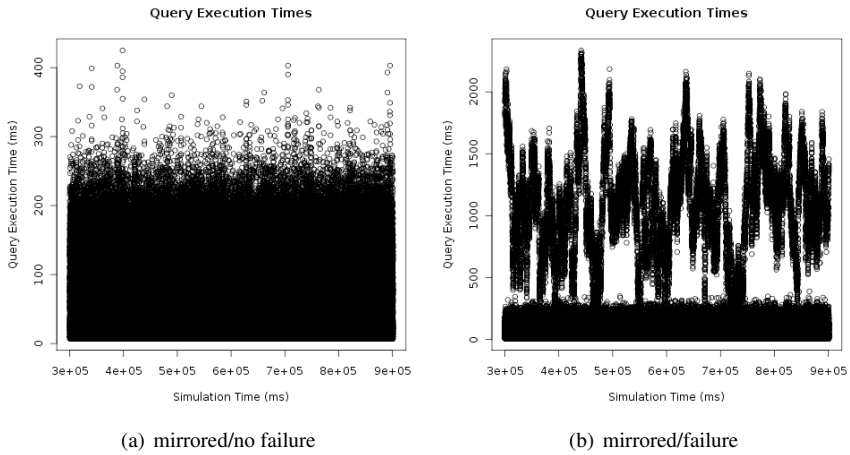


Figure 6: Simulated response times (ms) over simulation time (ms) for mirrored placement without failure(a), with injected failure (b)

Figure 6 shows the trace of query response times produced by our simulator using mirrored replica placement. During normal operations, as shown in Figure 6(a), the query load is distributed evenly among the nodes to which a tenant is assigned. Almost all queries are executed in less than 300 ms. Figure 6(b) shows the trace of a simulation run where 1 out of 20 nodes was failed at the beginning of the simulation. After the failure event, all queries were sent only to the remaining node until the failed node was unavailable. The simulation does not drop queries after a timeout but rather tries to process all queries to completion. As can be seen in Figure 6(b), the system does not recover from the 30 second failure injected at the beginning of the simulation using a mirrored configuration. While most of the queries are still executed in less than 300 ms, there is a considerable number of queries in the system which take up to 2 seconds to execute. This is a result of the two nodes affected by the failover trying to “catch up” with the query load. The load exceeds the capacity of the two nodea and therefore the amount of lost work cannot be regained. This leads to very high response times for the affected tenants, which can be seen in the scatterplot, which negatively affect the mean response time.

Figure 7 shows simulator response time traces based on the interleaved layout. Figure 7(a) shows the same behavior as the mirrored setup under normal operating conditions, therefore there is no inherent disadvantage to an interleaved setup. In fact, the mirrored configuration processes 359181 simulated queries with a 78 ms mean response time, while the interleaved setup processes slightly more queries (359577) with a slightly better mean response time of 72 ms. The improvement of the mean response time in the interleaved setup amounts to 8%. Although both numbers are not directly comparable, recall that a 7% improvement in throughput was observed in the real system under normal operating conditions when using an interleaved layout. As it can be seen in Figure 7(b), the injecting a failure has almost no impact for the interleaved configuration. The mean response time is 75 ms in spite of the failure, which is still lower than the mean response time in

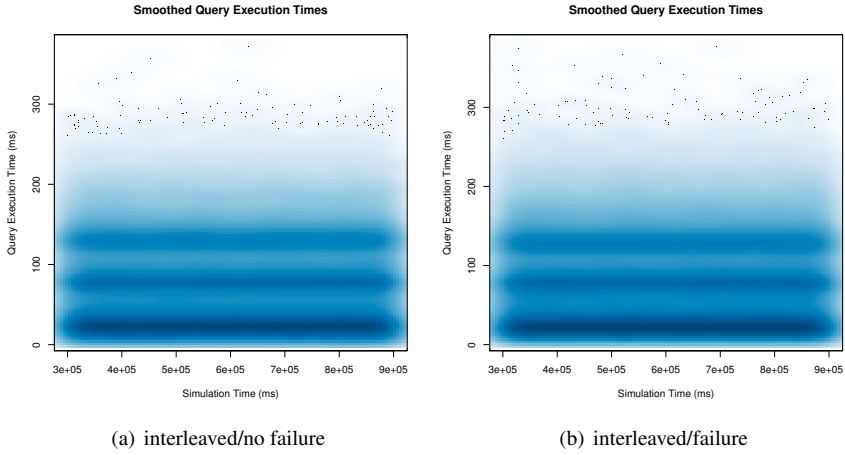


Figure 7: Simulated response times over time for interleaved placement without failure(a), with injected failure (b)

the non-failure mirrored configuration. The mean response time for the mirrored configuration with failure was 158. This 50% drop in performance is also consistent with our experiments on the EC2 cluster.

In this section, we have argued that the performance effects of deployment choices such as mirrored vs. interleaved replica placement can be shown using simulation. The simulation environment is however less complex than the real system. While changing the replication strategy in the real cluster is tedious to implement, changes to the simulator can be done much faster. Therefore, we see simulation as a tool for fast exploration of cluster configuration trade-offs, using which we are able to identify what configurations are worth implemented in our EC2 environment.

6 Related Work

Discrete event simulation has been successfully applied to many research areas in computer science [Cas05]. Discrete event simulation is based on the idea, that the observed system can be modeled as queuing network processes. Events created within or outside the system are based on a reference dataset. Therefore, reference datasets heavily influence the accuracy of the simulation.

Many standard tools supporting the design of such simulations have been made available to the research community, such as simulation languages (i.e. SimScript[KVMR75]) and simulation libraries like Simjava[HM98] or SimPy[Tea06] as used in our project. Frameworks and underlying techniques are continuously improved towards higher simulation accuracy and performance. In order to speedup the simulation methods for distributed and parallel discrete event simulations have been developed [FT96]. One actual

framework to be mentioned here is Parsec, which aims at parallel simulation of complex systems[BMT⁺98].

A wide variety of application specific simulators have been developed that incorporate the specifics of certain technologies, such as large-scale wired or wireless networks [ZBG98]. In the study of specific network-intensive workloads Saidi et al. “determine how accurately we can model the overall behavior of a complex system using appropriately tuned but relatively generic component models” [SBHR05], an approach we are trying to adapt for the modeling of the well-predictable in-memory execution components.

Also, widely distributed, job-oriented grid computing environments and the effects of scheduling on overall grid performance have been studied using simulation based on the modeling of applications [Cas02, BM02]. Many other examples exist, each showing that for the comparison of scenarios and the study of the impact of parameters simulation remains a verifiable complementing activity to empirical study.

Distributed systems and in particular Web server farms have previously been studied using simulation. Particularly similar to our method of studying in-memory database systems, which resemble dynamic web page generation in their CPU-bound nature, Teo studied the effect of load balancing strategies in clusters using simulation [TA01]. In this work, the client and Web server service times used in the simulator were also determined by carrying out a set of experiments on an actual testbed.

More specifically in the field of distributed database systems simulation has been used on a micro-level to study the performance of operators in a CPU-bound context [MD95], where it can also be seen that a CPU-bound application profile is always preferred in simulation, and that closed-loop systems always follow similar patterns in modeling user think times in their load model.

The data placement problem for relations has been previously studied in the context of parallel databases with large relations and a fixed cluster size. The Bubba parallel database prototype uses a statistics-driven variable declustering algorithm, that takes the access frequency and size of the relations into account [CABK88]. It therefore focuses on a single tenant placement problem within a fixed cluster of nodes and shows that load-balancing improves with increasing declustering. The prototyping of the Bubba system was supported by a simulation to “accurately predict the scalability of Bubba’s performance over the entire range of configuration sizes” [BAC⁺90]. A comprehensive simulation study of data placement in shared-nothing systems [MD97] has been conducted to find a consensus on the most efficient placement algorithm, following previous simulation studies specialized on data placement strategies such as multi-attribute declustering [GDQ92].

An autonomic and self-tuning cloud-based data warehousing framework has been described in our work on performance prediction[SEJ⁺ar]. By applying a load model to the entire cluster state, the framework can automatically conduct administrative actions on the cluster to optimize overall performance. Even though existing systems often contain self-management components that optimize threading, query admission and memory allocation, these systems do not consider data placement in a dynamically sized cluster in a multi-tenant context, where incremental re-organization is required rather than “big-bang” reorganization. Also, our research does not focus on distributing large relations, but

rather on heuristics for optimal redistribution of small relations. Also, the case for a cloud database service provider requires that optimization is not for minimizing response times but for maximizing utilization of resources under response-time constraints.

7 Conclusion

In this paper, we have presented the implementation of a simulation of a static cluster serving multiple tenants with analytic database services. The simulation results have been evaluated against the real system results and show that the simulator delivers adequate results for the evaluation of scenarios, such as failure conditions or overload. This simulation data can be used for system planning and design, for the detection of unexpected runtime behavior of real-life systems or to identify and validate hypotheses on multi-tenant database systems. Especially for the validation of proposed Service Level Agreements, simulation can compare many scenarios in parallel and compare the resulting economic benefit. For autonomic systems, a simulator can be used to train artificial intelligence algorithms, such as neural networks, in much shorter time and at lower cost, than using a real system. Also, a simulator has predictable runtime behavior that is not influenced by the measurement itself. Therefore simulation complements the empirical study of real systems in many useful ways. In our particular case, we could show that the interleaved placement performs much better than the mirrored placement when failures occur using a real experiment setup as well as using the simulation.

Future work on the simulation will involve the integration of cluster control in the Rock framework, with live system visualization and the simulation facilities. This might require the enhancement of the simulator to include on-line placement of tenants, cluster expansion, and memory resource management on an individual tenant basis. Additional simulation enhancements could include the simulation of merging the columnar data structures [KGT⁺10] using the simulated threading to study the impact of such maintenance tasks on the cluster performance, especially when considering the trade-off of losing excess capacity in the cluster vs performance improvements yielded by the merge. Another very interesting enhancement of the simulation would be to include the impact of disk I/O resource contention when using dynamic loading of inactive tenants to main memory or in situations where failures require re-loading the data from disk.

On the query simulation end, the support for delta-table performance impact simulation and its resulting write performance penalty because of queuing disk I/O for log-writing is regarded as future work, as is the impact of network communication overhead for multi-node joins.

Generally, the simulation evaluation component could be extended to apply the results to various SLA scenarios to calculate a profit or cost, which could be a basis to compare various configurations to each other on the basis of a single monetary figure. The dynamically adaptable cloud computing environment is especially suited for such a cost model, because the resources in the cluster have a clearly defined pricing based on their usage and the financial profile of each simulated scenario heavily depends on its computing resource

allocation and actual usage. The difficulty in such an assessment lies in the fact, that today's SaaS offerings usually define neither clear agreements in terms of clear boundaries for cases when the reliability of the service is insufficient (other than complete unavailability) nor any failure indemnification reimbursement policies which would make such an SLA-based loss-reduction calculation possible.

References

- [Ama] Amazon Elastic Compute Cloud (EC2). <http://aws.amazon.com/ec2>.
- [BAC⁺90] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, 1990.
- [BBG⁺95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, New York, NY, USA, 1995. ACM.
- [BM02] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, 2002.
- [BMT⁺98] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H.Y. Song. Parsec: A parallel simulation environment for complex systems. *Computer*, pages 77–85, 1998.
- [CABK88] G Copeland, W Alexander, E Boughter, and T Keller. Data placement in Bubba. *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 99–108, 1988.
- [Cas02] H. Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 430–437. IEEE, 2002.
- [Cas05] CG Cassandra. Discrete-Event Systems. *Handbook of networked and embedded control systems*, pages 71–89, 2005.
- [DGS⁺90] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.
- [DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, pages 205–220, 2007.
- [FT96] A Ferscha and SK Tripathi. Parallel and distributed simulation of discrete event systems. *Parallel and distributed computing handbook*, pages 1003–1041, 1996.
- [GDQ92] S Ghandeharizadeh, DJ DeWitt, and W Qureshi. A performance analysis of alternative multi-attribute declustering strategies. *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 29–38, 1992.

- [GHOS96] Jim Gray, Pat Helland, Patrick O’Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, New York, NY, USA, 1996. ACM.
- [HM98] F. Howell and R. McNab. simjava: a discrete event simulation package for Java with applications in computer systems modelling. In *Proceedings of the First International Conference on Web-based Modelling and Simulation*, 1998.
- [JA07] Dean Jacobs and Stefan Aulbach. Ruminations on Multi-Tenant Databases. In *BTW*, pages 514–521, 2007.
- [JLF10] Bernhard Jaecksch, Wolfgang Lehner, and Franz Faerber. A plan for OLAP. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *EDBT*, volume 426 of *ACM International Conference Proceeding Series*, pages 681–686. ACM, 2010.
- [KGT⁺10] Jens Krueger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Franz Faerber. Optimizing Write Performance for Read Optimized Databases. In *Database Systems for Advanced Applications, Japan*, 2010.
- [KVMR75] P.J. Kiviat, R. Villanueva, H.M. Markowitz, and E.C. Russell. *SIMSCRIPT II. 5 programming language*. CACI, 1975.
- [Lam98] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [LZGS84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.
- [MD95] M. Mehta and D.J. DeWitt. Managing intra-operator parallelism in parallel database systems. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 382–394. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1995.
- [MD97] M Mehta and DJ DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, 1997.
- [MS03] E Marcus and H Stern. *Blueprints for High Availability*. Wiley, 2003.
- [OOC07] P. E. O’Neil, E. J. O’Neil, and X. Chen. The Star Schema Benchmark (SSB), 2007. <http://www.cs.umb.edu/~poneil/StarSchemaB.PDF>.
- [Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 1–2. ACM, 2009.
- [PSKL02] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. TPC-DS, Taking Decision Support Benchmarking To The Next Level. In *SIGMOD ’02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 582–587, New York, NY, USA, 2002. ACM.
- [SAB⁺05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, Pat O’Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *Proceedings of the 31st international conference on very large data bases*, pages 553 – 564, 2005.

- [SBHR05] A.G. Saidi, N.L. Binkert, L.R. Hsu, and S.K. Reinhardt. Performance validation of network-intensive workloads on a full-system simulator. In *Proc. 2005 Workshop on Interaction between Operating System and Computer Architecture (IOSCA)*, pages 33–38. Citeseer, 2005.
- [SBKZ08] Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. A Hybrid Row-Column OLTP Database Architecture for Operational Reporting. In *BIRTE (Informal Proceedings)*, 2008.
- [SEJ⁺ar] J. Schaffner, B. Eckart, D. Jacobs, C. Schwarz, H. Plattner, and A. Zeier. Predicting In-Memory Database Performance for Automating Cluster Management Tasks. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, to appear.
- [SPvSA07] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of Caching and Replication Strategies for Web Applications. *IEEE Internet Computing*, 11(1):60–66, 2007.
- [Sto08] Michael Stonebraker. Technical perspective - One size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.
- [TA01] Y.M. Teo and R. Ayani. Comparison of load balancing strategies on cluster-based web servers. *SIMULATION-CALIFORNIA-*, 77(5/6):185–195, 2001.
- [Tea06] S.P.D. Team. Simpy homepage. <http://simpy.sourceforge.net/>, [Last accessed, 18(03):2007, 2006].
- [Ter85] DBC/1012 Database Computer System Manual Release 2. Teradata Corporation Document No. C10-0001-02, 1985.
- [TPC] TPC-H. <http://www.tpc.org/tpch/>.
- [ZBG98] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. *ACM SIGSIM Simulation Digest*, 28(1):154–161, 1998.

SSD \neq SSD – An Empirical Study to Identify Common Properties and Type-specific Behavior

Volker Hudlet, Daniel Schall
Databases and Information Systems Group
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
{hudlet, schall}@cs.uni-kl.de

Abstract: Solid-state disks are promising high access speed at low energy consumption. While the basic technology for SSDs – flash memory – is well established, new product models are constantly emerging. With each new SSD generation, their behavior pattern changes significantly and it is therefore difficult to make out characteristics for SSDs in general. In this paper, we accomplish empirical, database-centric performance measurements for SSDs, explain the results, and try to derive common characteristics. By comparing our measurement results, we detect no ground truth valid for all solid-state disks. Furthermore, we show that a number of prevalent assumptions about SSDs, which several SSD-specific DBMS optimizations are based on, are questionable by now. As a consequence of these findings, tailor-made DBMS algorithms for specific SSD types may be unsuitable and optimal use of SSD technology in an DBMS context may require careful design and rather adaptive algorithms.

1 Introduction

Solid-state disks gained a lot of attention lately. While many research papers present new algorithms to exploit performance of SSDs, enterprises are thinking of ways to incorporate them in their own products. Hence, flash chips and solid-state disks have become established products by now. SSDs offer dramatically improved random access behavior compared to conventional spinning disks. Due to their lack of mechanical parts, they are able to answer requests much faster than disks. That makes them perfect candidates for incorporating into database systems to speed up data processing. Due to the absence of spinning platters, SSDs also promise to come with a smaller energy footprint. Still, SSDs are not the swiss army knife of storage devices, they come with some limitations we have to deal with, for example read/write asymmetry. The market for solid-state disks is constantly changing and newer SSD generations are steadily improving their performance. With every new SSD generation, new product characteristics are emerging. Some drawbacks of earlier SSDs have been resolved in recent models. The constant change in the devices' properties makes it difficult for upper-layer algorithms to exploit the underlying storage. Optimizations tailored to a dedicated SSD model can have even negative effects on differently behaving models.

In this paper, we measure performance and energy consumption of five solid-state disks and compare them to manufacturer-provided data sheets. By pinpointing the specific characteristics of each drive, we claim that it is unwise for research to reveal the DB performance benefits of this disruptive technology change by relying on a single SSD type. We also derive common characteristics of flash drives in order to support research of the database community and leverage the exploration of flash-specific algorithms. This paper is structured as follows: In Section 2, we briefly summarize the internal structures of SSDs. Next, we outline important related work in Section 3. In Section 4, we are presenting our measurement methodology and the solid-state disks used for our performance study. Section 5 shows our experimental results. We interpret our results and compare them to the properties and facts listed in the data sheets of the manufacturers. In Section 6 we derive common characteristics and review some assumptions made about SSDs in research papers. In Section 7, we reflect our measurement setup and point out some possible limitations to our approach. Finally, we draw our conclusion in Section 8.

2 General SSD Characteristics

Solid-state disks are using flash chips for persistently storing data. An abstraction layer (*FTL*) on top of the chips provides a block device interface and hides the specific flash chip characteristics.

Flash Chips Flash chips are used to store persistent data on SSDs in a matrix of storage cells. The cells can either embody NOR or NAND gates. Modern cells, called *Multi-Level Cells* (MLC), can store more than one bit per cell. Today's solid-state disks are mostly composed of MLC NAND chips; we focus our description on that technology.

Reading from flash chips can only be done pagewise, where each page contains about 1 – 4 KB. It takes about 50 μs per page. Writing pages requires higher voltages, it takes about 10 times as long as reading ($\sim 500 \mu\text{s}$). Furthermore, each cell has to be erased prior to writing new values to it. Erasing is even more cost-intensive and can only be done in larger blocks, not by one page at a time. Typically, a block is 32 – 256 KB in size. It takes about 20 times as long to erase a block as reading a page ($\sim 1000 \mu\text{s}$). Erasing flash blocks leads to a slow but constant destruction of the cells. After about 10^5 erase cycles, cells will start to wear out and the block is no longer able to retain data.

Flash chips can be grouped together in so-called *planes* to increase storage capacity. Multiple planes can be accessed in parallel to enhance data throughput.

Flash Translation Layer To cope with the limitations of bare-metal flash chips, a mitigation layer is used on top of the chips/planes, called *Flash Translation Layer* (FTL). It provides a block-device interface to the upper layers, making the SSD look like a common storage disk. Therefore, the SSD user does not have to worry about erasure-before-write and handling worn-out blocks; these jobs are handled by the FTL. Because overwriting data on flash chips requires special treatment, the FTL must provide an erase-before-write mechanism that is able to save neighboring flash pages from being erased. Further, the FTL has to take care of worn-out cells.

Based on these basic functionalities, today's FTLs provide a lot more logic to further improve SSD performance. First of all, to avoid the need for erasing hot-spot areas over and over again, a page mapping is introduced to redirect logical page accesses to different physical locations on every new write. This helps to save erase cycles and, therefore, also improves performance. To free up unused areas of the SSD, a garbage collection schema can be implemented allowing the SSD to asynchronously perform erase and cleanup operations when the device is idle. Further reorganization tasks such as summarizing sparsely filled blocks can be employed. Like conventional hard disks, SSDs usually have an internal DRAM cache to buffer write requests or store prefetched pages. This buffer enables solid-state disks to backup and restore pages during erase cycles and to keep in-memory information, e.g., page-mapping structures. By using an FTL, it is possible to avoid most drawbacks of flash chips while making use of the advantages. Therefore, the FTL is a major performance-critical part of every SSD and manufacturers are eager to keep the implementation details a secret. A more detailed explanation of flash memory, SSDs, and their internal structures can be found in [RKM09, CPP⁺06].

3 Related Work

SSDs have been intensively explored in recent years with a focus on the characteristics of SSDs, on its integration into (existing) hardware systems and on its effective exploitation.

Operating Systems At the device level, Wang et al. [WKG09] propose non-in-place updates when writing to SSDs, which results in a performance improvement in their work, but which is automatically performed inside modern SSDs to mitigate wear-out. The same holds for flash-optimized file systems (e.g., *YAFFS* [Man02]) which employ journaled writes to avoid random and in-place updates. FTL implementation proposals [CPP⁺06, LPC⁺07, KKN⁺02] show different techniques that yield a logical-to-physical block mapping. As the FTL is embedded inside the device firmware, it is unknown whether any of these techniques or which particular technique has been adopted by the SSD manufacturers.

System Architecture Approaches concerning the system architecture try to find suitable solutions where to place SSDs in computer systems and how to incorporate them. Three main strategies [RKM09] are feasible using SSDs: as extended system memory, as storage accelerator, or as alternative storage device. Other works use a hybrid approach of SSDs and conventional HDDs where the SSD serves as persistent buffer for HDDs in order to mitigate I/O latency [CMB⁺10, KJKM09] or – depending on their workload – to adaptively place pages on one of these device types [KV08].

DBMS-specific Optimizations The exploitation of SSDs to increase the performance of data-intensive workloads still is in focus of the database systems community. An overview of the knobs and layers which can be made SSD-aware in a database system is given by Graefe [Gra09]. Of course, several components which interact and rely on external storage have been incorporated to be aligned to the characteristics of flash memory, so a lot of different proposals have been made in recent years. This includes amongst others SSD-

tailored DB buffer replacement algorithms [OHJ09], page layouts [THS⁺09], and index structures [AGS⁺09, KJKK07, WKC07]. Do et al. [DP09] show the impact of SSDs on join processing, mainly the tendency to become CPU-bound rather than being I/O-bound when HDDs are used. Tsirogiannis et al. [THS⁺09] recommend late materialization for speeding up join processing on SSDs.

SSD Measurements Papers in this area try to find out more about the SSD behavior and how to react to certain situations in reality. Most of them are based on micro-benchmarks used to reveal internal characteristics. Bouganim et al. [BJB09] and Chen et al. [CKZ09] were the first who tried to derive the intrinsic characteristics of different SSDs. They conclude that SSDs have to be considered as black boxes, as they follow no common rule. SSDs in RAID configurations have been examined in [BdNSS10] and [PABG10], where the latter one states that some effects of SSDs, e.g., the read-write asymmetry, are amplified due to the RAID mechanism. Overall, some characteristics are differently interpreted. [BJB09] state that they did not observe any performance improvements from submitting I/Os in parallel, [BdNSS10] use long queue depths and asynchronous I/O in order to increase the bandwidth.

As SSD advance, aspects covered by some approaches are already mitigated by the FTL. Others base their finding on a theoretical flash model, mostly by applying the metrics for read, write, and erase for raw flash chips or derive their results purely based on simulation. Even though theoretical effectiveness can be proven in this way, there can be quite a substantial discrepancy with regard to real-life SSDs. This can be seen in some papers, where the approaches are also verified on real SSDs, but the results are not as good as anticipated or derived by simulation. Moreover most papers base their experiments on only one type of SSD, neglecting the fact that their results could differ attributed to the employed SSD.

4 Methodology

For getting insights into SSD behavior, the read and write performance of the solid-state disks was measured. To stress all devices with the same access patterns, we developed a tool (similar to uFlip¹ and Iometer²) that allows us to perform benchmarks on the devices. The tool is able to read and write different access patterns from/to the devices. The page size the tool uses is adjustable. Figure 1 outlines the access patterns we used to benchmark the SSDs. The first test pattern is sequentially accessing n pages. This pattern is

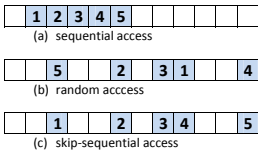


Figure 1: Access Patterns

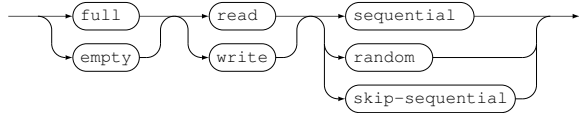


Figure 2: Measurement Combinations

¹<http://uflip.inria.fr/~uFLIP/>

²<http://www.iometer.org>

common in database servers when scanning through a table where no tailored access paths are available. Similarly, sequential writes are typical during log-flush operations. The second pattern is randomly accessing all pages of the test file. We ensured that each page gets accessed only once and that the pattern is repeatable. This pattern is often seen in databases when accessing pre-selected leaf pages in a B-tree. Random writes occur when updating several database tuples at a time or when flushing modified pages back to disk. The final pattern we tested is called *skip-sequential*, which accesses pages sequentially, but skips randomly over some pages. This pattern can be observed when a set of pages needs to be accessed, e.g., when – upon reading scattered database pages – the I/O requests are performed using ascending/descending physical addresses. Hence, this pattern evolves from the random pattern by pre-sorting the page numbers. This helps conventional disks to minimize head movement and, thus, reduces access times.

We set up a testing environment to benchmark all SSDs using the same hardware platform. To measure the device’s energy consumption, we attached the computer to a measurement device. This enables us to keep detailed track of the devices’ power consumption in addition to the performance measurements. A detailed description of the measurement setup can be found in [SHH10]. Using the previously described setup, we ran several tests against all SSDs and recorded their performance and energy consumption. The tests were run on a 1GB file filled with random data. We repeated the tests using varying page sizes and queue depths. First, we measured read and write patterns on a nearly empty drive (denoted by *empty* in Figure 2). After these measurements, we filled the drive with random data, while preserving the 1GB test file, and ran the same tests again. Figure 2 shows the benchmark combinations we ran for each device. We verified our results for sequential and random access by comparing it to results obtained by IOMeter and got more or less the same performance figures ($\pm 10\%$).

5 Experimental Results

In this section, we will present our measurement results for each SSD individually and discuss the observations. We measured using 32K pages for bandwidth and 2 – 8K pages for IOPS measurements.³ After we have shown all results, we will compare them and derive common patterns.

SSD1 – SuperTalent FSD32GC35M 32GB This SSD is the oldest one we tested. Results clearly show slow performance under all tested patterns as well as heavily degraded write performance. On the other hand, as the results show, random read is as fast as sequential read. SuperTalent states in the data sheet that this device can perform over 58,000 IOPS, a number we could not even get close to. Unfortunately, no further information about page sizes or queue depths is given.

SSD2 – Mtron MSP-SATA7525 The next SSD shows improved performance compared to SSD1, as depicted in Figure 5. Still, random writing is tremendously slower than other access methods. The SSD’s data sheet tells a lot more about the parameters used for

³According to [BJB09], 32KB is the preferable page size for SSDs.



Figure 3: Performance Measurements Using Different SSD Types

testing. It documents page sizes, queue depths, and access patterns used; therefore, the measurements are far more comprehensible. Though, we could not reproduce the stated performance, but get closer to it than we could for the first SSD we tested.

SSD3 – Intel X25-M G1 SSD3 represents the first Intel generation. As shown in Figure 5, this drive is really good at sequential reading, while random reading is comparatively slow. Then again, all write patterns are performing equally well. This is a significant operational difference compared to the first two SSDs. Intel’s data sheet documents the queue depth, but not the page size used for benchmarking sequential access patterns. For random accesses, the page size is mentioned. We were unable to get the same performance, even with the same parameters as in the data sheet.

SSD4 – Intel X25-M G2 The next generation of Intel SSDs came with the additional feature *TRIM support*⁴. Figure 5 indicates improved overall performance for all patterns. Nevertheless, the disk is showing the same challenges as the first generation. We could get closer to the performance reported in the data sheet, but were still unable to reach the advertised 35,000 IOPS. At least, we were able to measure the same sequential read bandwidth as stated (not shown in this figure).

SSD5 – Crucial RealSSD According to the manufacturer’s data sheet, the device can read up to 60,000 and write up to 45,000 pages/second. Our own measurements show quite a different picture. While reading on SSD5 is faster than on all other SSDs we tested, random writing stresses this device remarkably. Although the data sheet promises 60,000 IOPS for random read, we could not get even close to this number.

6 Results Interpretation

After we presented individual measurement results for each SSD, we are going to examine some common patterns observed on more than one device. In this section, we will also

⁴<http://t13.org/Documents/MinutesDefault.aspx?keyword=trim>

examine common assumptions regarding SSDs and show that not all of them are true.

Random Access Interestingly, though SSDs do not have moving parts and therefore should not suffer from random access, in fact, they do. Our measurements show that random access may be substantially slower than sequential access. Dependent on the device, this effect ranges from -5% performance on SSD1 and SSD2 up to -50% on SSD3 and SSD4. Therefore, sequential accesses should still be preferred over random accesses, although it is not as vital as on hard disks.

Considering the break-even point for selecting index-based access over sequential table scan, there is now a shift on SSDs. For conventional hard disks, a rule of thumb says to use an index-based scan only if the selectivity is below 1 - 3%, otherwise to scan the whole table sequentially. On solid-state disks, the selectivity factor can be shifted to higher percentages. Because of the different performance characteristics of solid-state disks, it is not possible to spot a clear break-even point. For example for SSD1 and SSD2, break-even would be at $\sim 90\%$, while on SSD3 and SSD4, break-even is at only $\sim 50\%$, due to their worse random access performance. On SSD5, break-even lies at $\sim 75\%$, thus a considerable divergence is visible for SSDs. Our results show that there is a trend towards faster write support on SSDs. Clearly, SSD1 and SSD2 suffer heavily from random writes, whereas the newer SSDs from Intel cope with them much better. SSD5, on the other hand, is again performing badly at random writing while providing fastest sequential writes.

Database query optimizers can decide between random and sequential access based on configurable disk parameters. Hence, the same care that has to be taken for conventional disks also has to be applied when using SSDs. Unfortunately, performance characteristics of SSDs are much harder to quantify than for spinning disks: On conventional disks, RPM, cache size, and bus delay are the only vital characteristics to estimate performance. On SSDs, there are no key performance indicators and characteristics can only be derived from measurements. The decision whether to write random pages (logically) in-place or to employ log-structured sequential writes strongly relies on the behavior of the underlying SSD. Therefore, optimizing algorithms for wrong device models can make overall performance even worse. Especially developers for flash-aware buffer algorithms have to consider that device-specific tweaks might be obsolete in no time.

Unstable Behavior While verifying the results using IOMeter, we observed another effect on SSD3. We did some changes to the source code of IOMeter to enable per-second tracking of measurement values. Using this tweaked version, we were able to get more detailed performance data from our devices. Figure 4 visualizes the write performance of SSD3 in pages/second on a per-second basis. As illustrated, every 4 to 5 seconds, performance is heavily degraded for about 3 seconds. We conclude, the drive is performing internal re-organization like freeing up flash blocks or searching for another writable block. We measured the same behavior for sequential writes, though the timespan between drop-offs was about 3 times longer. On SSD4 – the successor of SSD3 –, we measured similar behavior, although the performance drops during writes were not that severe.

While benchmarking SSD4, we had a look at the TRIM command introduced for this model and observed an interesting behavior. Figure 5 depicts our write-performance mea-

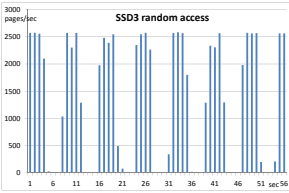


Figure 4: Iometer per Second

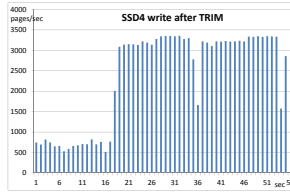


Figure 5: Write after TRIM

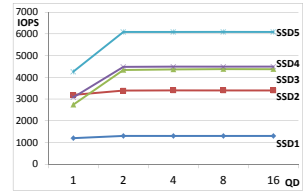


Figure 6: Queue Depth

surement right after deleting ~ 130 GB of files on the drive and issuing corresponding TRIM commands to the drive. In this graph, a heavily degraded performance in the first half of the measurement is evident. Apparently, the SSD tries to free up flash blocks while we were simultaneously applying a write load to it. The proprietary FTL mapping particularly concerns device caching, block allocation, and garbage collection. All these mechanisms are software controlled and entirely hidden to the upper software layers. Hence, optimization decisions in the OS or DBMS may be counterproductive and sometimes even worsen the time-consuming house-keeping activities. As inferred from Fig. 4, write latency may extremely vary. While less than $\sim 400 \mu\text{s}$ in the best case, we have observed outliers of more than some hundred ms, that is a device-dependent variance of more than $\sim 200 - 500$.⁵ Compared to that, a magnetic disk with a device-dependent variance of $\sim 2 - 5$ exhibits quite a stable access behavior and lends itself to reliable optimizer decisions.

Another aspect is a kind of heterogeneity among the SSD types present in a DBMS environment, where several heterogeneous SSDs may coexist in an application (or they may be dynamically exchanged). As a consequence, tailor-made algorithms for specific SSD types, e.g., concerning indexing or buffer management, are not very useful. The same arguments apply for specific workload optimizations (pure OLTP or OLAP processing, mixed workloads with varying degrees of reads/writes). A continuous adjustment or exchange of algorithms affected is not very practical in productive DBMS applications.

Read/Write Asymmetry As mentioned in the literature, reading flash pages is about 10 times faster than writing them because of intrinsic electrical properties. In conclusion, writing to SSDs should be equally slower than reading. Our measurements show that this is not true in general. SSD1 and SSD2, for example, do not exhibit degraded performance for (sequential) write, they are equally fast as sequential read. On all other SSDs, an asymmetry is measurable, but still not as bad as advertised.

Especially for buffer management algorithms, read/write asymmetries introduce a big potential for optimizations. On conventional disks, reading and writing cost the same; therefore, it does not make a big difference whether a clean or a dirty page gets evicted from the buffer. Considering solid-state disks, dependent on the device, the difference can now be significant. As mentioned earlier, current research papers already gave attention to this and flash-aware access algorithms were introduced. Nevertheless, it is crucial for these algorithms to know the exact properties of the underlying device, i. e., the precise read/write behavior, to enable optimizations.

⁵Note, we observed similar variance factors at the level of DBMS operations, e.g., splits in B*-trees, but these were provoked by algorithmic or implementation weaknesses.

Slower When Full? SSDs have to erase flash blocks prior to writing new values to it. As a consequence, overwriting some blocks on a full disk should be much slower than writing to an empty disk. We verified this assumption by filling all drives with random data and repeating our tests afterwards. No significant differences were measurable. Therefore, the common advice to leaving some empty space on the SSD to help the FTL find free/erasable pages can be abandoned. In fact, this is a waste of storage space, since our measurements do not indicate differences between empty and full drives.

Impact of Queue Depth As mentioned earlier in this paper, the *queue depth (QD)*, that is, the number of concurrent requests needing access to the device, can make a great difference to the overall performance. Due to a technique called *Native Command Queuing (NCQ)*, the device can re-order the sequence of requests in the queue to optimize its internal access path and improve throughput. This was primarily invented for hard disks to optimize their access path along the spinning platter. SSDs can still increase performance by optimizing switches between different flash planes. Furthermore, because access latency of SSDs is very low, bus delays gain greater influence in the overall access delay. To minimize communication overhead, higher queue depths in combination with NCQ can also be used to send bulk requests to the drive [Gas08]. This reduces the overhead to $1/bulk_size$ of the original overhead. SSD manufacturers know this fact and tweak their performance measurements accordingly. A high queue depth results in increased overall data throughput and higher IOPS, while a queue depth of 1 primarily minimizes access latency.

To gain more insights, we repeatedly measured various queue depths. By using a random read pattern, we give the FTLs a fair chance to optimize the queue. As Figure 6 indicates, the only significant improvement is between QD 1 and QD 2. Beyond this point, extending the QD did not improve data throughput. We did not expect this result, because manufacturers use even higher queue depths for their performance measurements. Also, current database servers do benefit from increased queue depths on conventional hard disks. As mentioned, some papers observed the same behavior [BJB09], while other papers explicitly recommend using longer queues [BdNSS10]. We see that it is not necessary to maintain long request queues for SSDs, thus database applications do not have to worry about getting maximum asynchronous I/O rates. A fair amount of outstanding requests is sufficient to keep an SSD at high bandwidth.

Energy Consumption As energy efficiency is getting a more and more critical factor for large data centers nowadays, we evaluated the SSDs' energy consumption. Figure 7(a) shows the absolute power consumption of the SSDs we tested. For this test, a sequential read pattern is used. Write patterns might consume even more energy. Obviously, the drives do consume energy when being idle; therefore, they are not as energy saving as expected. The SSDs' power profiles are similar to those of conventional hard disks, although their peak power consumption is considerably lower. Power consumption of conventional disks ranges from 4 – 6 Watts for mainstream disks to 9 – 14 Watts for enterprise server hardware. Figure 7(b) shows how many pages can be read by each SSD consuming one Joule of energy. As illustrated, *pages/Joule* are constantly rising, thus newer SSDs are getting more energy efficient. On conventional disks we measured only 600 – 1800 *pages/Joule*. Anyway, a more differentiated comparison is cumbersome, because

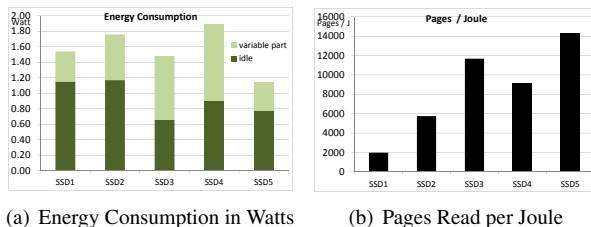


Figure 7: Energy Consumption Measurements

of the different performance characteristics and their implications on energy efficiency. In general, the best performing SSDs are still the most energy efficient.

7 Limitations

We explored the performance of I/O patterns typically occurring in a DBMS environment. However, we were and still are not aware of the tricks and assumptions, e.g., massive I/O parallelism, of the manufacturers under which their performance behavior was achieved as reported in the data sheets. Our results sometimes indicate substantial deviations confirming that the “promised” device behavior may not be fully exploited by DB applications. Although we took care that our measurements are accurate and reliable, there might be some limitations to our approach. In order to keep up fair publishing policies, we do not want to hide them from the readers. We mainly focused on two ideas, why our measurements might not be 100% reliable.

SSD Choice The solid-state disks we tested were not fresh out-of-the-box, but were rather used by our research group for several benchmarks previously. Ranging from SSD1, which is approximately 3 years old, to our recently bought SSD5, all drives were used in varying degrees. Therefore, the observations and claims we made in this paper might only be true for our particular devices. Since the older drives (SSD1 to SSD3) do not support the TRIM command to free up flash pages, these devices might be worn out exceptionally. Due to a limited budget, we tested only one device for every model; therefore, we cannot not make assumptions for whole product models, but rather for single product instances.

Measurement Platform Choice The original use for our measurement track was to measure and optimize energy consumption. For this reason, we only used a small server board for testing, which might not have a high-performance SATA bus controller. This might bottleneck our measurements. In order to eliminate this possibility, we verified our measurements using a dedicated SAS controller card. Nevertheless, the results stayed the same and we were unable to get the performance promised in the data sheets. We even switched the entire hardware to a different system, which did not lead to an improvement. We can not resolve all doubts for sure, but we assume our measurements were correct and there is in fact a major gap between manufacturer’s data sheets and real-world performance.

8 Conclusion

As the measurements clearly discover, each SSD exhibits a differing performance profile. We were able to identify some common patterns and outlined areas that are improving continuously, e.g., write performance. Still, manufacturers' claims about their drives' performance can not be blindly trusted. Because a common benchmarking procedure does not exist, performance claims of data sheets can hardly be reproduced in real-world scenarios. Due to the advancement of the internal FTL, larger write caches, and TRIM support, we believe that the often mentioned drawbacks of SSDs, e.g., slow writes, will soon disappear. Also, write endurance of SSDs is constantly rising and we do no longer have to care about destroying the disk by constantly writing to it. As we have proven by our experiments, every drive has its own characteristics. Optimization towards a single drive or against flash-chip access characteristics is no longer suitable under these circumstances. A lot of literature focuses on improved algorithms for flash chips, although there are no bare-metal flash chips in server systems. As we have shown in Section 6, current solid-state disks embody unpredictable behavior and performance may sometimes drop unexpectedly. In order to use SSDs in time-critical operations, like meeting deadlines in a real-time DBMS, algorithms have to be aware of these characteristics to anticipate even worst-case situations. More generic algorithms – not adjusted to single SSD types, but able to handle a widespread of different device characteristics – would be better suited and rather eligible for DBMS use. To suit a specific device, its characteristics could be determined either offline – prior to using the device in a productive environment – or online – during use. Then, according to the measured properties, the algorithm could automatically tune itself to maximize its performance. We propose that this approach would be more sustainable, even over SSD generations with changing behavior and be, therefore, more useful than highly specialized algorithms fitting particular SSDs only. Our benchmarks of today's solid-state disks unveiled a lot of pitfalls, although these measurements are far from being complete. For example, focusing on smaller grained, longer running benchmarks could help identify a lot more peculiarities of SSDs. For example, long running stress tests could reveal resource exhaustion inside the FTL or, by cutting power to the SSDs, persistence tests could be performed. Nearly all of the measurements we ran have discovered another fact for SSDs; therefore, we think there is a lot more to detect. Hopefully, future SSD generations will no longer need special treatment by the upper layers, because FTLs will contain more and more logic. We propose that SSDs will soon unite the advantages of conventional hard disks combined with faster random access behavior.

References

- [AGS⁺09] Devesh Agrawal, Deepak Ganesan, Ramesh K. Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *PVLDB*, 2(1):361–372, 2009.
- [BdNSS10] Stephan Baumann, Giel de Nijs, Michael Strobel, and Kai-Uwe Sattler. Flashing Databases: Expectations and Limitations. In *DaMoN*, 2010.

- [BJB09] Luc Bouganim, Björn Thór Jónsson, and Philippe Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.
- [CKZ09] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS/Performance*, pages 181–192, 2009.
- [CMB⁺10] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD Bufferpool Extensions for Database Systems. *PVLDB*, 3(2):1435–1446, 2010.
- [CPP⁺06] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. System Software for Flash Memory: A Survey. In *EUC*, pages 394–404, 2006.
- [DP09] Jaeyoung Do and Jignesh M. Patel. Join Processing for Flash SSDs: Remembering Past Lessons. In *DaMoN*, pages 1–8, 2009.
- [Gas08] Geoff Gasior. Intel’s X25-E Extreme Solid-state Drive. Technical report, The Tech Report, 2008.
- [Gra09] Goetz Graefe. The Five-Minute Rule 20 Years Later (and How Flash Memory Changes the Rules). *Commun. ACM*, 52(7):48–59, 2009.
- [KJKK07] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. mu-Tree : An Ordered Index Structure for NAND Flash. In *EMSOFT*, pages 144–153, 2007.
- [KJKM09] S.-H. Kim, D. Jung, J.-S. Kim, and S. Maeng. HeteroDrive: Re-shaping the storage access pattern of OLTP workload using SSD. In *IWSSPS*, pages 13–17, 2009.
- [KKN⁺02] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient Flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48:366–375, 2002.
- [KV08] Ioannis Koltsidas and Stratis Viglas. Flashing up the storage layer. *PVLDB*, 1(1):514–525, 2008.
- [LPC⁺07] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *TECS*, 6(3), 2007.
- [Man02] Charles Manning. YAFFS (Yet Another Flash File System) <http://www.yaffs.net/>, 2002.
- [OHJ09] Yi Ou, Theo Härder, and Peiquan Jin. CFDC: A Flash-aware Replacement Policy for Database Buffer Management. In *DaMoN*, pages 15–20, 2009.
- [PABG10] Ilia Petrov, Guillermo G. Almeida, Alejandro P. Buchmann, and Ulrich Gräf. Building Large Storage Based On Flash Disks. In *ADMS*, 2010.
- [RKM09] David Roberts, Taeho Kgil, and Trevor N. Mudge. Integrating NAND flash devices onto servers. *Commun. ACM*, 52(4):98–103, 2009.
- [SHH10] Daniel Schall, Volker Hudlet, and Theo Härder. Enhancing Energy Efficiency of Database Applications Using SSDs. In *C3S2E*, pages 1–9, 2010.
- [THS⁺09] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query Processing Techniques for Solid State Drives. In *SIGMOD*, pages 59–72, 2009.
- [WKG09] Yongkun Wang, Kazuo Goda, and Masaru Kitsuregawa. Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems. In *DEXA*, pages 777–791, 2009.
- [WKC07] Chin-Hsien Wu, Tei-Wei Kuo, and Li-Ping Chang. An efficient B-tree layer implementation for flash-memory storage systems. *TECS*, 6(3), 2007.

HiSim: A Highly Extensible Large-Scale P2P Network Simulator

Lukas Rupprecht

Jessica Smejkal
Alfons Kemper

Angelika Reiser

Fakultät für Informatik, Technische Universität München
Firstname.Lastname@in.tum.de

Abstract: The popularity of Peer-to-Peer networks is increasing rapidly but developing new protocols for P2P systems is a very complex task as testing and evaluating distributed systems involves high effort. P2P simulators are being developed to tackle this difficulty, to reduce cost and to speed up development. We describe HiSim, a modular and highly scalable P2P network simulator based on the simulation framework PeerSim which we use to simulate HiSbase, a P2P framework for efficient processing of multidimensional data. Because of its modular design, HiSim can easily be extended, e.g., to fit other application domains. Basic design problems, related to query processing, are introduced in general and concretely solved within HiSim. Additionally, HiSim provides mechanisms to evaluate new protocols using an integrated statistics component. We demonstrate the high scalability by performing simulations of up to $2 \cdot 10^4$ peers and $2 \cdot 10^7$ queries.

1 Introduction

The popularity of Peer-to-Peer (P2P) networks is increasing rapidly [RD10]. Not only filesharing protocols like, e.g., BitTorrent¹ but also the science community is utilizing them heavily. *E-science* communities form data grids to process huge amounts of data (e.g. the Large Hadron Collider at Cern²) by combining their available resources using P2P technology.

Developing new mechanisms and protocols for P2P systems is a very complex task as it requires a distributed testing infrastructure to verify and evaluate the protocols. Such an infrastructure should be as realistic as possible but, especially in the case of large-scale P2P networks, this is hard to achieve as costs are increasing rapidly with higher numbers of peers. The distributed testing process itself is very time-consuming and complex because each system needs to be monitored individually and the gathered data needs to be combined and analyzed. Additionally, evaluating new protocols for a certain task implies a complete distributed implementation of these protocols. Sometimes this is too costly or not possible due to time limits, especially, if these protocols are only prototypes which should only give a first impression if the desired approach is working.

¹www.bittorrent.com

²lhc.web.cern.ch/lhc

To overcome these problems, P2P network simulators are used. A simulator can run on only one single workstation and provide a central interface to attach and detach protocols easily. It allows faster and more clear testing as the whole P2P network runs centralized and the testing data does not need to be combined from several peers. In this paper we present HiSim, a highly scalable and easily extensible P2P network simulator based on the PeerSim simulation engine [JMJV], which realizes all the above mentioned advantages. We use the simulator to evaluate *HiSbase*, a P2P framework for managing and processing multidimensional E-science data. However, because of its modular design, the simulator can easily be extended to fit arbitrary application domains. The rest of the paper is structured as follows. Section 2 introduces the basics of HiSbase. Then the design of HiSim is explained. In section 4 the extensibility of the simulator is demonstrated by some examples. The next section provides a scalability analysis and sections 6 and 7 give a short overview of other simulator projects and summarize the main aspects.

2 HiSbase

HiSbase is the P2P framework for which the simulator was built. The framework provides functionality for data partitioning of multidimensional data and efficient query processing with the focus on preserving data locality and processing range queries [SBG⁺07], [SBM⁺09].

HiSbase is built upon the distributed hash table overlay structure *Pastry* [RD01]. *Pastry* coordinates the communication between the peers (or *nodes*) and provides a routing mechanism. It uses a one-dimensional ring topology where data and peers are mapped to. Additionally, *Pastry* optimizes the routing by implementing a proximity neighbor selection algorithm [CDHR03] which prefers physical neighbors when routing messages.

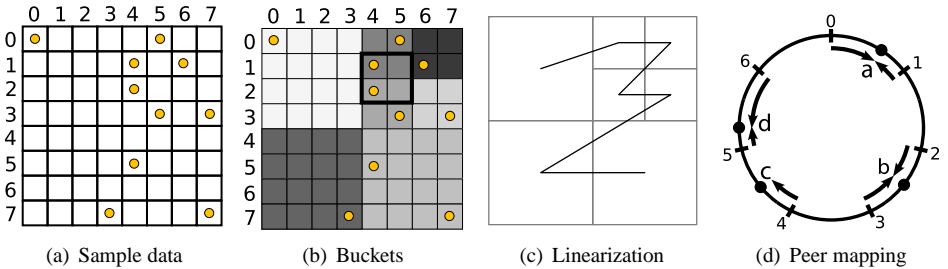


Figure 1: A sample for data placement in HiSbase [SBG⁺07].

HiSbase deploys histograms to partition multidimensional data and to create buckets containing approximately the same number of data elements (see Figure 1(b)). It partitions the data recursively, using a quadtree, and maps the leaves (or *regions*) to the overlay ring topology, using the Z-order (see Figure 1(c)). The combination of quadtrees and space-filling curves handles data skew and preserves data locality when mapping n-dimensional data to the 1-dimensional ring. Peers and regions are mapped to the key space by using

a hash function and regions are assigned to peers by their closest distance to peers (see Figure 1(d)). These histograms enable efficient query processing of region-based queries [SRK09]. Therefore a coordinating region is chosen from all regions covered by the query. A special message is routed to the node (the *coordinator*) responsible for this particular region. The coordinator then sends messages to all queried regions, and the responsible nodes look up the data in their local databases. They send the data back to the coordinator which combines all answers and sends the complete data back to the query initiator. As the space-filling curve preserves locality, it is very likely that only few nodes are responsible for the whole area covered by the query and that these nodes are neighbors in the ring.

3 The Simulator

We now introduce HiSim. We briefly present the deployed PeerSim API followed by an overview of problems occurring, when simulating large-scale P2P networks. We provide solutions to these problems and explain the main design of the simulator. HiSim is designed for HiSbase where we use it to conduct query simulations, create large-scale networks, evaluate new protocols and gather statistics on network load distribution.

3.1 PeerSim

PeerSim is a P2P simulation environment, implemented in Java [MJ09]. Its simulation engine provides the basic functionality for simulating and managing network connections and passing messages. A network is represented as a list of nodes, each of them maintaining a list of protocol objects. Additionally, initializer objects (executed before the simulation) and control objects (passive simulation monitors executed periodically) exist. PeerSim supports two main simulation approaches, cycle-driven and event-driven. In the cyclic model, protocols are executed periodically while in the event-driven case, protocol execution is triggered by messages which are sent via the simulated transport layer. The simulation is controlled by a configuration file which offers an easy way to set the different

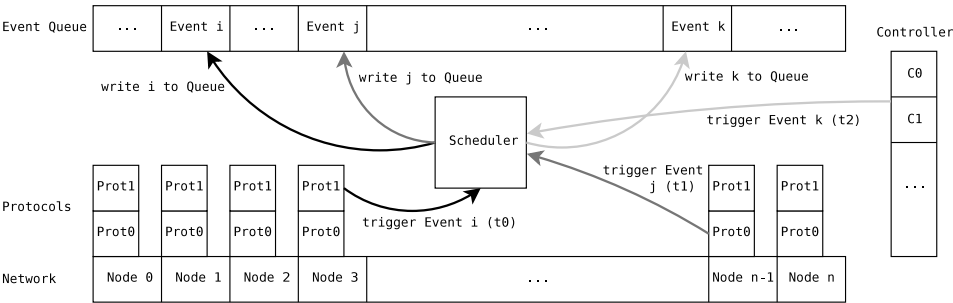


Figure 2: A sample execution of a PeerSim simulation.

simulation parameters. The PeerSim engine itself is single-threaded and all events (periodic and non-periodic) are written to one central event queue. The execution order of the events is determined by a scheduler which schedules the events according to their internal simulator time t (see Figure 2; isochromatic arrows indicate that the corresponding actions are performed in one atomic step). We decided to use PeerSim as its scalability outnumbers other common used simulators [NLB⁺07] and its API is well-designed and provides a good basis for extensibility. Additionally it is in wide use by the research community (see [JMJV] for a list of publications).

3.2 Design Issues for Large-Scale P2P Networks

When designing a P2P simulator architecture, usually two basic design issues arise: the choice between cycle- and event-based simulations and between single-threaded and completely parallel (one thread per node) execution.

Using a cycle- or an event-driven simulation approach heavily depends on the application. A cyclic engine can be used to simulate gossiping mechanisms, swarm intelligence techniques or to gather simulation statistics periodically. Event-driven scenarios involve the simulation of messages or queries. PeerSim provides engines for both cases and even the possibility to combine them. We use the event-based engine for simulating queries and apply the cyclic model to periodically gather statistics on the network load (see section 4.2). We deploy the combined approach in one protocol which processes queries and performs periodic checks on a node's load (see section 4.1).

The choice between single- or multi-threaded takes several aspects into account. The single-threaded approach (as implemented in PeerSim) has certain advantages compared to the multi-threaded one. A sequential simulator needs no scheduling by the operating system and can be executed on standard single-core processors which limits the network size only to main memory [MJ09]. Multi-threaded simulators on the other hand need massive scheduling efforts or large high performance computing clusters to run. Otherwise, the scheduling effort grows with the networks and limits scalability. Additionally, no parallelization techniques are needed when running a single-threaded simulator which makes development and evaluation much easier. One disadvantage, however, is that sequential simulators can not achieve true realistic behavior concerning aspects like throughput measures or modeling database lookups, as the particular nodes are not autonomous. Additionally, they run on only one single processor and hence, can not utilize the full available power of common multi-core systems. Besides these two separate approaches, one could also think of a hybrid approach, combining the features of single-threaded and multi-threaded simulators. Such simulators could deploy the existing computational power by running a certain number of peers on all available processor cores in parallel. Hence, each core executes its peers sequentially, the scheduling effort stays low as the different cores are running real parallel and multi-core architectures can be used to boost the simulator scalability. This approach could also be applied to HiSim. However, developing such simulators is much more complex than developing single-threaded simulators because hybrid simulators need to be thread-safe and the particular cores have to be synchronized. Making

HiSim a hybrid simulator is subject to future work as we only present the basic simulator here. In the following section we list some problems associated with single-threaded simulations and propose solutions.

3.3 Achieve Realism in Single-Threaded Simulator Environments

In a P2P system like HiSbase a high ratio of processing time depends on the database lookup of queried data. After receiving a query, a node looks up the queried data in its database. While the other nodes keep on processing, this node waits until the lookup has been completed before sending the corresponding answer. In a parallel simulation, we can stop the thread for a certain time to easily model the lookup. In a single-threaded environment, we can not keep a node waiting as this would interrupt the whole simulation, i.e. all other nodes. To solve this problem we manipulate the PeerSim event queue. Note that in the following we omit network latencies and message delays caused by physical factors due to clarity reasons. Assume that the lookup for query i to node k is scheduled at position p_1 in the event queue. The submission of the corresponding answer is scheduled at position p_2 . Then, $p_2 = p_1 + q$ with q being the number of events, scheduled between p_1 and p_2 (see Figure 3(a)). Note that these q events can be any kind of events triggered by any node or control object and have nothing to do with the query and thus, do not model a delay. They only lie between p_1 and p_2 because we operate in a single-threaded environment. In

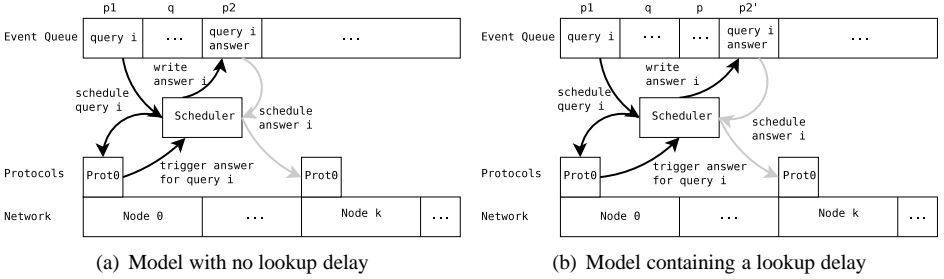


Figure 3: Modeling query lookup delays in single-threaded environments.

a parallel simulation, they would be processed simultaneously by different threads. Figure 3(b) now shows how we model a delay. We simply add a value d to the time, when the answer message would originally be scheduled. This postpones the message to the back of the event queue by p positions and hence $p_2' = p_1 + q + p$ holds. Consequently, the message arrives later at its target which represents our desired lookup. d can either be chosen as a constant or determined dynamically according to the amount of data the query demands.

Another problem occurs if load aspects are analyzed by the simulator. In HiSbase, each node maintains a FIFO query queue where incoming queries are listed. Currently processed queries are removed. The load of a node is defined by the number of queries, waiting in this queue. Transferring this load definition to the simulator is not possible due

to the single-threaded environment. A query comes in, is processed and the answer is submitted, all in one computational step which means that during that time no other actions can be performed by any other node and the whole simulator. Thus, no other queries can queue at the node while it processes a query. As a result, the load of a node would at most be 1. This situation is shown in Figure 4(a) (recall, that isochromatic arrows represent atomic simulation steps). To overcome this problem we could scan the simulator's

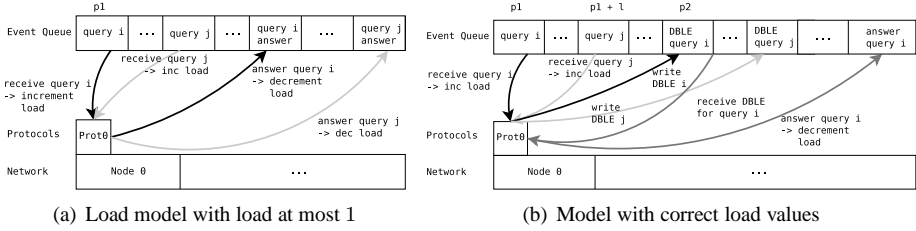


Figure 4: Modeling load in single-threaded environments.

event queue, looking for queries which have been sent to a node and have already arrived at their target. The resulting count would be the load. As the event queue can grow very large for huge networks, this is no efficient solution. Our solution is to introduce a DBLookupEvent (DBLE). Assume that a node's query request is scheduled at position p_1 in the event queue. After receiving the request the node sends a DBLE to itself (scheduled by the PeerSim Scheduler at p_2) but does not answer the query yet. The answer to the query is not submitted until the corresponding DBLE is received. The node can now queue additional query requests, scheduled at positions between p_1 and p_2 which makes load values > 1 possible (see Figure 4(b)). This load now conforms to our load definition. Note that the DBLE event models the time, a query waits in a node's query queue to be processed. The actual lookup delay is realized with the above described mechanism. Hence, the number of DBLE events scheduled for a node corresponds to the load of this node and the simulator time between p_1 and p_2 represents the time, the query waits in the queue.

3.4 Modular Architecture

Besides extending the simulation framework with aspects of query evaluation and load, our simulator is also characterized by its easy extensibility. PeerSim itself provides a solid basis for extensibility but for the HiSbase application and to keep our simulator generic, we added some new features.

For P2P systems built on top of overlay topologies, the question arises, which one to choose. Protocol behavior might differ on various overlay structures and the most suitable needs to be determined. To simplify this task, we provide the OverlayProtocol interface (see Figure 5(a)). This interface partially coincides with the KBR (key-based routing) API, proposed in [DZD⁺03]. This API comprises several operations which should be pro-

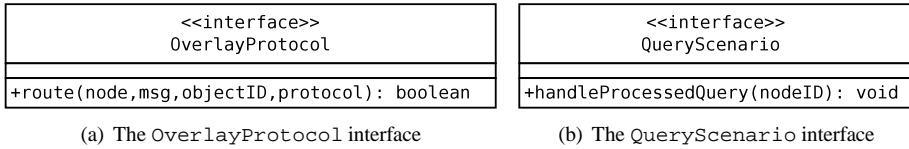


Figure 5: Interfaces for overlay networks and query simulations.

vided by structured overlays, including a `route` operation. In HiSim, overlay topologies can be implemented using this interface which requires each overlay protocol to provide a `route()` method for message routing. Top-level protocols can be built on this generic overlay protocol by just deploying the generic `route()` method from the underlying overlay and hence, overlays can be exchanged simply by specifying them via the configuration file. As HiSbase is built upon Pastry, HiSim completely simulates the FreePastry³ layer, including join and routing mechanisms.

One major part of HiSbase is its query processing functionality. Thus, the query simulation engine is an essential part of HiSim. When simulating queries, different scenarios are possible. A query scenario defines, how a node reacts after it received the answer to a previously sent query. One scenario might be that a node waits a random time period after receiving an answer and then submits a random number of queries. Another imaginable scenario is, when all nodes have batch-jobs of n queries (maximum load scenario) which they submit to the network always keeping m parallel queries in the network (*MPL m*). After receiving an answer, a node immediately submits a new query to the network to keep its m queries within the network. With our `QueryScenario` interface we provide an easy way to add new scenarios and to switch between them. Scenarios are added by implementing this interface and nodes only call `handleProcessedQuery` after they finished processing a query. The scenario then reacts accordingly. HiSim provides an implementation of a maximum load scenario and a cyclic scenario where in each cycle, a random number of queries is submitted from a random node.

4 Extensibility of HiSim

Different aspects can be thought of, when it comes to the extensibility of a given simulator. For HiSim, we show how easily new protocols can be added, statistics can be collected and additional functionalities, like storing and loading networks, can be realized.

4.1 A Sample Protocol for Dynamic Replication in HiSbase

As a first application, the simulator is used to evaluate a dynamic replication protocol (DRP) for HiSbase. Basically, the DRP works as follows. A node's current load is moni-

³<http://www.freepastry.org/FreePastry/>

tored periodically and if it increases above a specified level, it replicates parts of its data, placing it on other nodes determined according to their interest in the data and their load. If the load decreases again, the node revokes its replicas.

The DRP requires an extended query processing mechanism as it needs to take care of replicas. This mechanism differs from the standard processing mechanism in HiSbase and hence, we encapsulate it as one dedicated protocol. With the above described interfaces we can easily place it on top of our Pastry implementation and run it with the already implemented query scenarios without the need of further adaption. As stated above, protocols can combine the event- and cycle-driven approaches. We use this feature for the dynamic replication to perform the periodic checks on a node's load status and the query processing in the same protocol. As a result we receive a completely autonomous protocol, selectable via the configuration file and runnable instantly with different query simulation scenarios on top of Pastry. Other protocols can be added in the same way which obviously is fast and easy and keeps the design of the simulator clean. As a next task, we want to evaluate the DRP by gathering statistics from the network.

4.2 Recording Simulator Statistics

For evaluating, comparing, and analyzing different protocols, monitoring and recording statistics from the network is inevitable. We introduce a statistics component, implemented as a control object (see section 2) which periodically collects network statistics. We provide statistics on the total network load, on single node loads and on the number of queries, submitted to a HiSbase histogram region. The output format of the statistics files are tab separated values which allow convenient further processing, e.g., by using gnuplot⁴. Figures 6(a) and 6(b) show examples of such plots and were created by just passing the output files of the statistics observer to gnuplot. Adding further statistics as

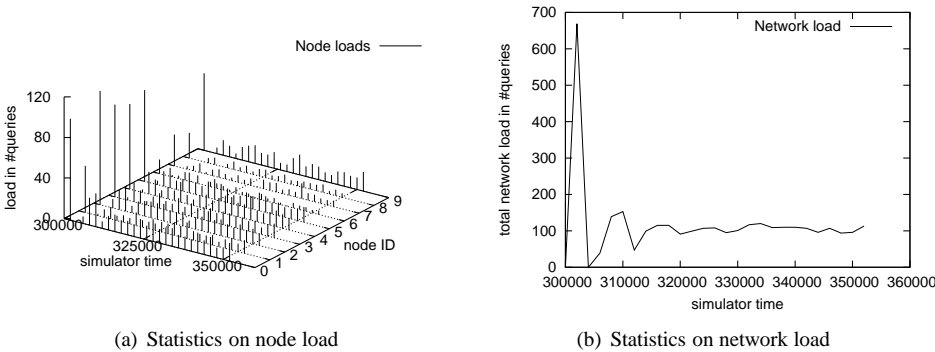


Figure 6: Sample statistics output.

e.g. a message counter, involves only low additional effort.

⁴<http://www.gnuplot.info/>

4.3 Saving and Loading Existing Network Configurations

As it is computationally intensive to create large-scale networks, our simulator provides a store and load mechanism for already created network configurations. Each simulation consists of a creation phase p_c and a simulation phase p_s . In p_s , the main simulation part, the query processing, is performed while in p_c , the network is built by using the join protocol of the specified overlay structure. When developing and evaluating new protocols it is inevitable to execute runs with the exact same characteristics as the results should be reproducible and comparable. With the store and load functionality, the state of a network (consisting of all node states) after p_c and before p_s is retained and can be reused later. Especially for large-scale networks of 10^4 or more nodes, saving a network configuration is very useful as creation time increases exponentially (see also section 5) and just reading network configurations is more than 90% faster.

5 Scalability Analysis

To analyze the scalability we performed two types of measurements. At first we measured how long network creation takes for different network sizes. We then simulated maximum load scenarios (see section 3.4) with various numbers of queries, using the previously created networks. All measures were performed on a workstation equipped with two Intel Xeon quadcore processors (2.93 Ghz) and 64 GB RAM.

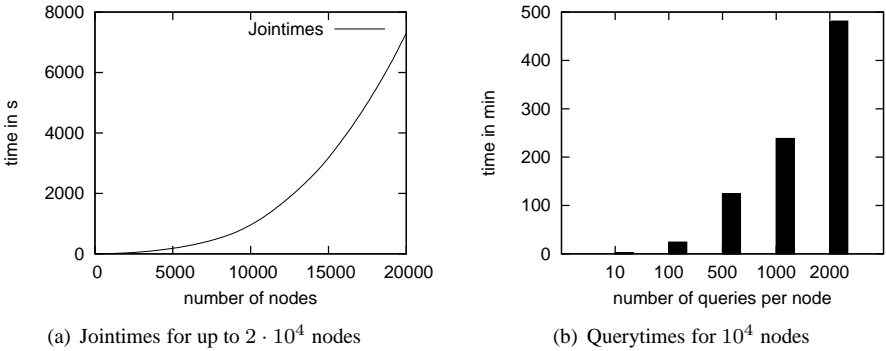


Figure 7: Scalability results.

Figure 7(a) shows the times it takes to create a network of $2 \cdot 10^4$ nodes. Times were recorded each time 100 new nodes joined the network. We see that these times increase exponentially with the number of nodes. This is due to the fact that the more nodes exist, the more messages a node needs to send to join the network. Thus, creating a Pastry network of size 10^4 with HiSim takes about 15 minutes while $2 \cdot 10^4$ nodes already take about 2 hours. With our store-and-load functionality this effort only has to be taken once and is thus reduced to a minimum.

To simulate the query batch-jobs, we used a query set of $2 \cdot 10^4$ queries, collected from log files on a real database, storing two-dimensional astrophysical data. We conducted runs with 10, 100, 500, 1000 and 2000 queries per node. The queries were picked randomly and submitted to the network of 10^4 nodes with an MPL of 10 (for the 10 and 100 queries per node scenarios) resp. 100 (for the rest). The results in Figure 7(b) show the durations of the complete simulations. They span from 2.3 minutes for 10 queries per node to 481 minutes for 2000 queries. Looking at the results, we observe that the query simulations scale approximately linearly in the number of queries. The same setup was carried out with $2 \cdot 10^4$ nodes and the results confirm the linear scalability.

The performance measures show that we can simulate large-scale networks with high query load within a reasonable time period. The simulation durations are especially tolerable as we do not have the resources to set up such systems in reality. Additionally, evaluating different new protocols is much easier when operating on a single workstation. As a result we save a lot of time by deploying our simulator for these tasks.

6 Related Work

Many different network simulators exist to investigate P2P systems. HiSbase itself already has an integrated simulator which uses the FreePastry simulation engine. This engine allows simulating FreePastry code without any adaption but the engine is very lightweight as delays, message drops etc. are not modeled. Additionally, as the simulator is multi-threaded, network sizes of only up to 10^3 nodes are possible which brought up the need for a new simulator. Other simulator examples are NS-2 [Ber], NeuroGrid [Jos03] or GPS [YAg05] to mention only a few. NS-2 is a widely used simulator which simulates the network layer on packet-level. This is useful to analyze networks on lower layers but comes with the loss of scalability. It can be used on multiple machines and runs in parallel. GPS and NeuroGrid are two single-threaded simulation engines. NeuroGrid was initially designed for comparative simulations between Freenet-, Gnutella- and NeuroGrid-based systems. GPS is implemented in Java and completely driven by messages. No cyclic protocols are supported.

According to [NLB⁺07] most of the published research in P2P systems, conducted with the help of simulators, is based on custom software. This software does not deploy a standard API like the ones mentioned above or simply does not mention the underlying simulator architecture. P2PRealm [KVK⁺06], for example, is a simulator especially designed for studying neural network algorithms. With HiSim, we present a simulator which is built on a common basis and provides the extensibility to fit special applications.

7 Summary and Future Work

Our work was driven by the need of a P2P network simulator which can be easily extended and provides high scalability to implement and evaluate new protocols for HiSbase, a P2P

framework for handling multidimensional data. The presented HiSim meets these requirements. Its design allows adding and exchanging new protocols easily and thus, HiSim can also be deployed in other domains. It is based on PeerSim, a highly scalable P2P simulation framework which provides a single-threaded simulation engine. HiSim utilizes PeerSim to manage low level tasks like passing messages and manage physical connections. Beyond that, HiSim provides functionality especially designed for query processing applications built on top of overlay networks. The introduced design allows convenient management of various query scenarios which can easily be executed on different overlay structures. We showed that single-threaded engines are arbitrarily scalable in theory, because no thread limits caused by the operating system apply, but that other problems arise from this fact. The key idea to solve these problems is to manipulate the simulator's event queue. HiSim presents solutions to query-related problems occurring in single-threaded environments and can thus provide more realistic query behavior. With some examples we demonstrated the possibilities to extend and adapt HiSim to fit many additional tasks. We analyzed the scalability by running simulations with different node and query numbers. The results show that network creation time increases exponentially but we can reduce that effort to a minimum with the provided store-and-load mechanism. Additionally we demonstrated that the query simulation time scales linearly in the number of queries.

Future work will include more specific analyses which will compare real time system runs and simulation runs. These tests will help to determine optimal parameter settings for the query delay and for the lookup delay. We will also add new protocols and evaluate them to improve and extend the HiSbase system itself. Another major task is to equip HiSim with multi-core support.

References

- [Ber] Berkeley/LNBL/ISI. The NS-2 Network Simulator. <http://www.isi.edu/nsnam/ns/>.
- [CDHR03] M. Castro, P. Druschel, Y.C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. In *Future Directions in Distributed Computing*, pages 103–107. Springer-Verlag, 2003.
- [DZD⁺03] F. Dabek, B. Zhao, P. Druschel, J. Kubiawicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. *Peer-to-Peer Systems II*, pages 33–44, 2003.
- [JMJV] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim Simulator. <http://peersim.sf.net>.
- [Jos03] S. Joseph. An extendible open source P2P simulator. *P2P Journal*, 1:1–15, 2003.
- [KVK⁺06] N. Kotilainen, M. Vapa, T. Keltanen, A. Auvinen, and J. Vuori. P2PRealm - peer-to-peer network simulator. In *2006 11th International Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks*, pages 93–99, 2006.
- [MJ09] A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer (P2P'09)*, pages 99–100, 2009.

- [NLB⁺07] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *ACM SIGCOMM Computer Communication Review*, 37(2):98, 2007.
- [RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, volume 11, pages 329–350, 2001.
- [RD10] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, 2010.
- [SBG⁺07] T. Scholl, B. Bauer, B. Gufler, R. Kuntschke, D. Weber, A. Reiser, and A. Kemper. HiSbase: histogram-based P2P main memory data management. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1394–1397. VLDB Endowment, 2007.
- [SBM⁺09] T. Scholl, B. Bauer, J. Müller, B. Gufler, A. Reiser, and A. Kemper. Workload-aware data partitioning in community-driven data grids. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 36–47. ACM, 2009.
- [SRK09] T. Scholl, A. Reiser, and A. Kemper. Collaborative query coordination in community-driven data grids. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 197–206. ACM, 2009.
- [YAg05] Weishuai Yang and Nael Abu-gazaleh. GPS: a general peer-to-peer simulator and its use for modeling BitTorrent. In *Proc. IEEE/ACM MASCOTS05*, pages 425–432, 2005.

Operators for Analyzing and Modifying Probabilistic Data – A Question of Efficiency

Jochen Adamek, Katrin Eisenreich, Volker Markl, Philipp Rösch

j.adamek@tu-berlin.de, katrin.eisenreich@sap.com,
volker.markl@tu-berlin.de, philipp.roesch@sap.com

Abstract: To enable analyses and decision support over historic, forecast, and estimated data, efficient querying and modification of probabilistic data is an important aspect. In earlier work, we proposed a data model and operators for the analysis and the modification of uncertain data in support of what-if scenario analysis. Naturally, and as discussed broadly in previous research, the representation of uncertain data introduces additional complexity to queries over such data. When targeting the interactive creation and evaluation of scenarios, we must be aware of the run-time performance of the provided functionalities in order to better estimate response times and reveal potentials for optimizations to users. The present paper builds on our previous work, addressing both a comprehensive evaluation of the complexity of selected operators as well as an experimental validation. Specifically, we investigate effects of varying operator parameterizations and the underlying data characteristics. We provide examples in the context of a simple analysis process and discuss our findings and possible optimizations.

1 Introduction

In the decision making process, we need to consider risks and chances of future developments. To this end, the derivation and evaluation of scenarios based on different assumptions about the future is a powerful technique. However, as per definition, applying an assumption always introduces uncertainty to the data at hand. This uncertainty must be appropriately represented in the data. Existing uncertainty management approaches mostly address applications in the fields of scientific and sensor data processing, spatial databases, information extraction, or data cleansing. Decision support over large volumes of data including both uncertain data and certain information, e.g., from a data warehouse, has received comparatively little attention so far. A prominent exception is the work presented in [JXW⁺08], which relies completely on a sample-first approach. In previous work (see [ERM⁺10]), we in contrast apply a model-extension approach to represent, analyze, and modify uncertain data. Our primary goal is to support users in the flexible creation and evaluation of what-if scenarios over (partially) uncertain data represented through arbitrary distributions. We consider the process of what-if analysis as an iteration of steps of data analysis and scenario creation as described in [ER10]. Apart from its iterative nature, we also point out that we aim to enable users to conduct the analysis process in a highly interactive fashion. In the best case, a user should be able to derive a scenario, analyze it,

change some of its underlying data, and analyze the resulting alternative scenario within seconds. Naturally, the additional complexity implied by the representation of uncertainty poses a major challenge when aiming at low response times. A comprehensive analysis of run-times is therefore a major contribution when assessing the overall performance and general applicability of our approach. Moreover, it can serve us to discover room for improvements.

To exemplify the application scope of our solution, consider the following use cases:

- Use case UC 1: An analyst wants to prospect next year’s revenue in a newly developed region R_{new} . He takes the past development of a similar region R_{ref} as reference for his prediction. Additionally, he wants to take into account available forecasts about the general economic development.
- Use case UC 2: A user analyzes the process of delivery and deployment of orders. He wants to investigate possible resource costs caused by deployment personnel during a specific time frame. To this end, he applies different assumptions regarding temporally uncertain delivery times and deployment durations.

As noted above, most of the existing approaches for uncertain data management focus on efficient querying and analyses of (mainly discrete) probabilistic information. The aspect of modifying data to create scenarios, which is a central aspect of our work, is mostly out of their scope. In the remainder of the paper, we will therefore foremost discuss those operators, yet emphasize that the ‘traditional’ aspect of data analysis is also an integral part of our approach. We briefly describe important aspects of our data model and an operator set for deriving and converting uncertain values, as well as analysis and modification of such values in Section 2. We then evaluate the complexity of the selected set of operators (Section 3) and provide an experimental validation of their costs based on a prototypical implementation (Section 4). We address opportunities for optimizing the computation of steps in an analysis process in Section 5. We present related work in Section 6 and summarize our findings in Section 7.

2 Data Model and Functionalities

What-if analyses and decision support over uncertain data require a flexible data model and powerful operators which both are introduced in our previous work, see [ER10, ERM⁺10]. Our data model allows for the use of both symbolic and histogram-based representations of uncertainty, similar to [SMM⁺08]. An uncertain value x_i ¹ is associated with a distribution P_i which can be represented symbolically or as a histogram \bar{P}_i . A histogram comprises β_i bins $B_i = \{b_1, \dots, b_{\beta_i}\}$ within a lower support (boundary) l_i and an upper support (boundary) h_i . Each bin $b_j \in B_i$ is associated with a density w_j . Similarly, we use two-dimensional histograms to represent two-dimensional distributions $P_{x,y}$. The discussion in this work is focused on the usage of equi-width histograms (EWH). Alternative

¹Where possible, we omit the subscript for reasons of readability.

partitioning schemes are conceivable but imply higher costs and are out of the scope of the current work.

By investigating typical questions for what-if analyses and processes implementing such analyses (see [ER10]), we identified a set of operations we deem specifically important in this context as discussed in the following.

Introducing and converting uncertain information First, at the start of an analysis process, no specific knowledge about a value’s distribution is given in many cases. Users therefore may want to derive information about an (expected) distribution from historic fact data. For example, in UC 1 the user assumes next year’s revenue development in region R_{new} to follow the distribution observed in a similar region R_{ref} . To introduce this information, he derives and stores a histogram over last year’s recorded revenue values for cities in R_{ref} by using our operator DRV . Another basic operation (CNV) serves to change the representation of such derived or externally provided uncertain values. This step can be applied explicitly, e.g., when users want to convert a symbolic into a histogram-based representation as basis for flexible modifications. Moreover, it can be adopted implicitly, e.g., when an input is given in symbolic form but the executed operator requires a histogram-based representation. The derivation and conversion of the predicted (relative) economic growth inc_e and the (relative) regional revenue increase inc_{rev} are schematically depicted in the left portion of Figure 1 and further discussed in Sections 3.1 and 3.2.

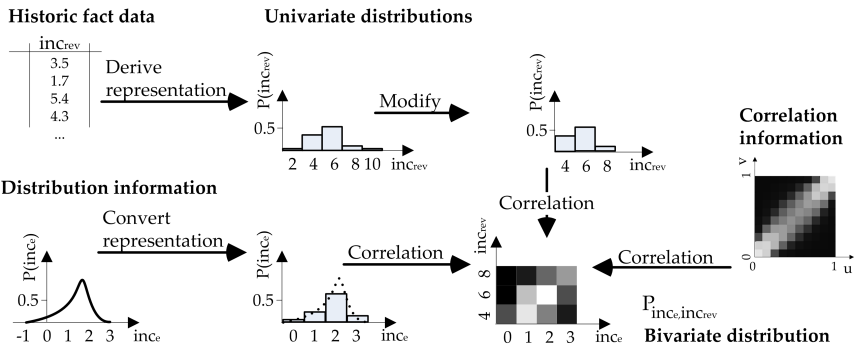


Figure 1: Overview of exemplary representation and processing of distributions

Evaluating the uncertain information Once (uncertain) data is available in appropriate representations, users need analysis capabilities such as computing aggregates, or selecting and viewing values based on a provided threshold. While such data analysis functionality is a natural part of our approach (contributing to the analysis part of the what-if analysis process), it is not the focus of this paper. Rather, we mainly concentrate on (i) the modification of uncertain data, (ii) the handling of dependency in data, and (iii) the issue of temporal indeterminacy, all of which are particularly estimable aspects in the context of scenario-based planning:

Modifying uncertain data: The modification of uncertain data is essential to support the application of assumptions about the development of selected aspects to derive scenarios representing potential future states of the world. For example, considering the inc_{rev} value in UC 1, a user might want to discard “outlier” information or apply the assumption that extremely low revenues will be avoided due to appropriate marketing measures. This is reflected by modifying (*MOD*) part of the distribution of inc_{rev} , as depicted in the middle portion of Figure 1. In other cases, users want to update an old value altogether to assume a new distribution for a future value. In both cases, we can view the applied modification as a creation of a new scenario. To retain the relation between values in different scenarios, we do not replace the original value but store the modification result together with a reference to the original value. When modifying histograms, we use a delta approach, that is, we keep the (bin-wise) delta of the new value to the old value. Besides enabling users to investigate the lineage of values, the delta approach enables more efficient incorporation of modification results in further processing steps (see Sections 3.3 and 5).

Introducing dependencies: Unlike many other approaches for probabilistic data management, we explicitly target the representation and processing of arbitrary correlation structures between uncertain values. Specifically, we address the case where a user wants to *introduce* an assumed correlation between two values when no dependency information can be faithfully derived. This can be the case, e.g., when underlying fact data is too sparse or when two values are provided as independent data. The latter occurs in UC 1, where we dispose of separate values for the forecast economic development (inc_e) and the derived revenue increase inc_{rev} . To model arbitrary forms of correlation (e.g., linear correlation or high dependencies of extreme values) we apply the approach of copula functions as described in [ER10]. Rather than constructing copulas at run-time (implying calls to statistical functions and corresponding costs) we precompute, store, and process them in the form of a histogram-based approximative correlation representation (ACR). The introduction of correlation is depicted in the right-hand portion of Figure 1, where two univariate distributions ($P_{inc_{rev}}$ and P_{inc_e}) are combined into a joint distribution based on correlation information from an ACR. Details of the ACR approach are discussed further in Section 3.4.

Temporal uncertainty in planning processes: Finally, in addition to representing and processing uncertainty of (measure) values, we also enable the representation of temporal indeterminacy, i.e., uncertainty in the temporal allocation of data [ERM⁺10]. For example, use case UC 2 rises the necessity to compute costs for the deployment of ordered products following their uncertain delivery times and assuming an uncertain duration of deployment. In this context, we consider the handling of indeterminate “events” occurring during some uncertain time interval. In Section 3.5 we consider the aggregation (AGG^T) over measures associated with such events within a time interval \underline{T} .

For a more detailed description of the set of operators and their application in the decision support context, we refer to [ER10, ERM⁺10].

3 Analytical Evaluation

In this section, we analyze the complexity of the operators lined out above under different parameterizations and varying input data characteristics. In general, except for the case of converting representations, we assume input values to be represented in the histogram-based form; i.e., we do not consider the application of our operators for modification, correlation introduction, and temporal aggregation on symbolic representations. In case a distribution is given in symbolic form, we can use the conversion operator to yield the respective histogram-based representation. Consequently, the number of bins β used to represent univariate distributions is the most relevant factor as regards the complexity of the evaluated operators. Further, for the derivation of distributions, the number of underlying facts (n_F) is crucial. Similarly, where sampling is applied, the number of sample elements (n_S) naturally influences not only the "accuracy" of results but also the implied costs for their derivation. When processing or constructing symbolic representations of distributions, costs for computing specific statistics, such as quantiles, can vary. We do not consider such distribution- and implementation-dependent costs in this section, but rather examine them experimentally in Section 4.

3.1 Derivation of Representations

We support the derivation of distributions over values of fact data, which users can then use as a basic input for their analyses. The intuition is that such a distribution may constitute a proper reference for the development of another value in some similar context. For example, in our use case UC 1, the analyst wants to utilize knowledge about the past revenue increases in region R_{ref} as reference for the prospective revenues in a newly developed region R_{new} with no historic data available. To this end, he needs to construct a distribution from the historic revenues of all stores in R_{ref} .

The $DRV(F, tgt)$ operator serves exactly this purpose, essentially enabling the *introduction* of uncertainty based on fact data. It receives a number n_F of facts from a column in the fact table F of our database. The target distribution P is specified via the tgt parameter, including the representation type and further parameters determining P . In particular, the representation form can be either histogram-based or symbolic. In the former case, a user must further provide the number of bins β and the lower and upper support (l, h) of the desired histogram \bar{P} . In the latter case, a user must provide the assumed function of the distribution based on some insight or expectation about the underlying facts.

Histogram Representation When deriving a histogram over the fact values, we assume they are provided in a non-sorted order. In the case of equi-width histograms, which we focus in this paper, we statically compute bin boundaries based on the desired lower and upper support (l, h) and the number of bins β and assign each of the n_F values, resulting in complexity $O(n_F)$. In the general case, for each of the n_F values underlying the histogram to-be derived, we apply a bisection algorithm for sorting it into one of the β bins of the

target histogram, resulting in a complexity of $O(n_F \cdot \log_2(\beta))$.

Symbolic Representation In our prototype, we support the derivation of uniform, Gaussian, and Gamma distributions, while further functions could be added. Finding the lower and upper bounds of the distribution support of an assumed uniform distribution requires one scan of the underlying facts to determine their maximum and minimum. For a Gaussian, we similarly need to process each of the n_F values to iteratively compute the mean and variance. We currently estimate the scale and shape parameters of a Gamma distribution from those parameters. Hence, for all considered distribution functions, the computation implies a complexity of $O(n_F)$. The specific costs for a given target function naturally depend on the individual calculations conducted over each fact value.

3.2 Conversion of Representations

The operator $CNV(x, tgt)$ allows users to flexibly change the way of managing the uncertain data, converting the symbolic distribution representation of a value x to a histogram and vice versa, depending on the parameters specifying the target distribution tgt . Further, users can change the resolution of a histogram, e.g., for statistical error analysis, by setting a new number of bins β . Returning to UC 1, a user wants to incorporate information about the economic forecast for the region R_{new} . This forecast is provided by means of an expected value and an associated confidence interval and is represented in the system as a Gaussian with appropriate mean and variance. In order to further process this value, the user converts it to a histogram-based form. Another application of CNV arises when users want to test a (derived) distribution for goodness of fit with actual data; such tests (i.e., the χ^2 -test) often rely on binned data.

Similar to the DRV operator, users must provide parameters specifying the desired distribution tgt , including the type of representation and representation-specific parameters. The three potential cases of conversion are as follows:

Symbolic Distribution into Histogram For constructing a histogram from a given distribution function, such as a Gaussian, the user defines the lower and upper support (l, h) of the target histogram and the desired number of bins β or, alternatively, an optimal β can be estimated using a basic heuristic aiming at an optimal approximation of the interval with β bins (e.g., Sturges rule). For each of the β bins, the source distribution is integrated within the lower and upper bin boundary, implying a complexity of $O(\beta)$. In the case of a uniform source distribution, density values for β bins based on l and h equally results in $O(\beta)$.

Histogram into Symbolic Distribution To compute parameters of an assumed uniform distribution from a histogram we need to find its lower and upper bounds by reading β bins or, if available, exploit stored metadata about l and h , inducing a complexity of $O(\beta)$ or a constant access cost $O(1)$, respectively. To estimate parameters of a Gaussian or Gamma

distribution, we compute the required parameters (mean and variance or scale and shape, respectively) through an iterative run over all bins, inducing a complexity of $O(\beta)$.

Histogram into Histogram As a third alternative, we can convert a source histogram \bar{P}_x with β_x bins into a new histogram \bar{P}_y with β_y bins. This conversion serves, e.g., to provide users with a changed granularity of information or to ensure two histograms have the same bin resolution. This might be required for a succeeding operation such as testing the fit of two distributions to each other. The conversion proceeds by finding the bin boundaries of the β_y bins of \bar{P}_y and computing the area covered by the β_x bins in \bar{P}_x within the boundaries of each bin of \bar{P}_y . The imposed complexity of the computation is $O(\beta_x + \beta_y)$.

3.3 Modification of Uncertainty

Modification of an uncertain value is necessary to introduce a new assumption about its concrete distribution in a potential scenario or adapt its value otherwise. For example, in UC 1, the derived distribution for the (relative) regional revenue increase inc_{rev} might include large tails due to outlier data. If the user assumes those tails of the distribution irrelevant for his current analysis (or does not want to consider this part of the distribution in his scenario), he can modify $\bar{P}_{inc_{rev}}$ by setting the densities associated with relevant bins of $\bar{P}_{inc_{rev}}$ to 0, as illustrated in Figure 1. In UC 2, the analyst can modify expected deployment start times of selected orders to analyze the potential influence on the resulting deployment costs within the time slot under consideration.

The operator $MOD(x_{old}, x_{new}, cond)$ is applied to histograms $\bar{P}_{x_{old}}$ and $\bar{P}_{x_{new}}$ to change the represented distributions. A modification causes the frequencies associated with selected bins of $\bar{P}_{x_{old}}$ to be changed, optionally depending on a specified condition $cond$. This way, a user can explicitly specify both the affected bins and their target density through x_{new} ; alternatively, he can provide a condition for determining the bins whose density shall be changed as well as their new density value. An example is the application of a predicate to modify a certain part (e.g., the tail) of a distribution by specifying a condition on the (new) lower and upper support of x_{new} . To ensure that modified values can be traced back to the original value, we do not replace x_{old} but insert a new value x_{new} with a reference to x_{old} . This way, we can further apply and compare multiple modifications. Physically, the modification is written back as the delta \bar{P}_Δ (bin-wise difference) from $\bar{P}_{x_{old}}$ to $\bar{P}_{x_{new}}$. A worst case complexity of $O(\beta)$ is induced by reading β bins and writing delta density values for all bins. The influence on actual costs depends on the resulting degree of modification (f_m), i.e., the fraction to which a distribution is actually affected by a conditional modification.

3.4 Introduction of Correlation

The possibility to introduce correlation to previously independent (or independently represented) values is a valuable means to investigate effects of dependencies between values in data, e.g., to analyze the probability of extreme values occurring jointly. In UC 1, we assume that the user wants to evaluate a correlation among values that were provided separately; they are represented by the revenue increase distribution $P_{inc_{rev}}$ and the forecast economic growth P_{inc_e} .

We use the operator $COR(x, y, H, d)$ to enable the introduction of a correlation structure determined by H and d between two distributions P_x and P_y . As noted before, the processing of COR is based on the usage of copulas. In brief, a copula C is a distribution function representing the relation between two marginals F and G and their joint distribution H . The formal foundation is Sklar's Theorem [Skl59, MST07], which states that, given H as a bivariate² distribution with F and G as univariate marginal distributions, there exists a (copula) function $C : [0, 1]^2 \rightarrow [0, 1]$ so that $H(x, y) = C(F(x), G(y))$. Using the inversion approach (see, e.g., [Nel06]), we can construct a copula $C(u, v) = H(F^{-1}(u), G^{-1}(v))$, u and v being uniforms over $[0, 1]$. We write $C_{H,d}$ to denote a copula where the structure of the represented correlation is determined by the distribution H and the correlation degree d . To correlate two arbitrary distributions P_x and P_y , we then again apply Sklar's Theorem, substituting F and G with the desired marginals P_x and P_y . We investigate both the complexity of the native (sampling-based) approach and our approach based on approximate correlation representations (ACRs).

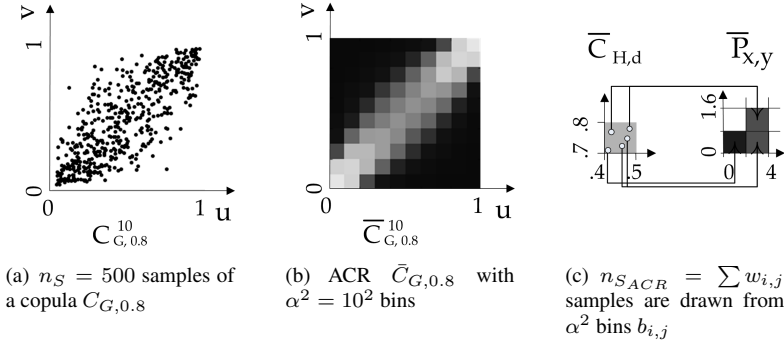


Figure 2: Factors influencing the performance of correlation introduction

Sampling-based copula approach For the sampling-based approach, we must consider the costs for both constructing and applying the copula. We first draw n_S samples of its underlying bivariate distribution H with correlation d . We transform the samples using the cumulative distribution function of each of F and G to construct the copula as distribution over $[0, 1]^2$. See Figure 2(a) for an example of a Gaussian copula $C_{G,0.8}$ with 500 samples.

²Without loss of generality, we restrict our considerations to the bivariate case, i.e., only consider the correlation amongst two variables.

The copula construction requires $3 \cdot n_S$ computations applying calls to statistical functions, inducing a complexity of $O(n_S)$. Applying the constructed copula implies, for the two coordinates of each of the n_S samples, a computation of the quantiles of the distributions P_x and P_y (provided as histograms \bar{P}_x and \bar{P}_y). The computation of quantiles requires a binary search over the frequency values of the β_x, β_y bins of \bar{P}_x and \bar{P}_y succeeded by computing the concrete distribution within the bin based on a uniform spread assumption. In total, the sampling-based approach – including the construction and application of the copula – results in a complexity of $O(n_S + n_S \log_2(\beta_x) + n_S \log_2(\beta_y))$.

ACR-based approach Aiming at lower response times and an independence from statistical library calls at run-time, we pre-compute copulas $C_{H,d}$ and store them as ACRs with α^2 bins, denoted by $\bar{C}_{H,d}^\alpha$ (see Figure 2(b)). Then, to correlate two values, the system chooses and applies an appropriate ACR based on the desired correlation parameters. This way, no costs for copula construction are induced at run-time. Rather, we use the aggregated information stored in the ACR, i.e., the coordinates of the α^2 bins $b_{i,j}$ and their respective weights (densities) $w_{i,j}$. As discussed in [ER10], we decrease the negative influence of discretization by applying inversion based on artificial samples from each bin $b_{i,j}$. This is indicated for an individual bin in Figure 2(c). Those samples are uniformly distributed within each $b_{i,j}$, the sample number per bin being relative to its weight $w_{i,j}$. The correlation introduction then proceeds exactly as for the sampling-based approach. Using a total of $n_{S_{ACR}}$ samples over all bins, this implies a complexity of $O(n_{S_{ACR}} \log_2(\beta_x) + n_{S_{ACR}} \log_2(\beta_y))$ for computing quantiles of P_x and P_y for each sample. In the usual case, we consider $n_{S_{ACR}}$ to be similar to or higher than n_S to ensure comparable result accuracy. A very basic approach is to invert the coordinates of the α^2 bin centers only (instead of inverting $n_{S_{ACR}}$ sample coordinates), resulting in a complexity of $O(\alpha^2 \log_2(\beta_x) + \alpha^2 \log_2(\beta_y))$. In a usual case, α^2 is a magnitude smaller than $n_{S_{ACR}}$, resulting in, e.g., $\alpha^2 = 40^2 = 1600$ rather than $n_{S_{ACR}} = 40000$ quantile computations for P_x and P_y . However, those savings come at the cost of mostly unacceptable discretization errors.

3.5 Indeterminate Temporal Aggregation

In order to enable the handling of temporal indeterminacy of plans, we consider the analysis over indeterminate events. The indeterminacy of an event e_i is reflected through an uncertain start time t_i and a duration d_i . As an example, to implement UC 2, we need to compute the prospective overall deployment costs implied by a number of indeterminate deployments $e_i \in E_{dep}$ during a specified interval, e.g., $\underline{T} = [1, 5]$. We use the operator $AGG^{\underline{T}}(X, E, \underline{T})$ to compute the aggregate (sum, minimum, or maximum) of values of an attribute $X = \{x_0, \dots, x_n\}$ associated with temporally indeterminate events $E = \{e_1, \dots, e_n\}$ within a time interval $\underline{T} = [l_{\underline{T}}, h_{\underline{T}}]$. To compute the aggregate, we must consider all events that have a potential overlap with \underline{T} (i.e., $l_{t_i} < h_{\underline{T}} \wedge h_{t_i} + h_{d_i} > l_{\underline{T}}$). In the following, we consider the aggregation over a single event $e \in E$ (omitting the subscript for reasons of readability). Essentially, we need to

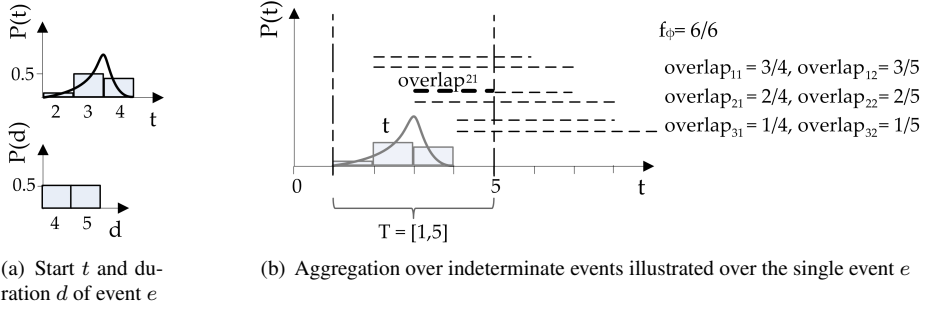


Figure 3: Representation and processing of temporal indeterminacy associated with indeterminate events

compute the fraction ϕ to yield the contribution $\phi \cdot x$ of the measure value x associated with e . The fraction ϕ depends both on the position of \underline{T} and on the set of all possible intervals $I_{pq} \in \mathcal{I} = \{[t_p^{start} = v_p, t_{pq}^{end} = v_p + v_q]\}$, $v_p \in I_t, v_q \in I_d$ in which e can occur. To give an example, Figure 3(a) depicts the start time t , $\beta_t = 3$ and duration d , $\beta_d = 2$ of the event e while Figure 3(b) shows how aggregation works over this single event. In Figure 3(b), the fraction ϕ of e results from six possible occurrence intervals $\mathcal{I} = \{I_{11}, I_{12}, \dots, I_{32}\}$. For each possible interval, we need to compute joint probabilities $P(t = v_p) \cdot P(d = v_q)$ (assuming independence between t and d) and the fractions of overlaps, i.e., the part of I_{pq} that lies within $\underline{T} = [1, 5]$.

We denote the average number of potential occurrence intervals of all considered events $e_i \in E$ as $n_I = (\sum_{i=0 \dots N} \beta_{t_i} \cdot \beta_{d_i}) / |E|$ and the fraction of those intervals that actually overlap \underline{T} (and therefore contribute to the aggregate result) as f_ϕ . Temporal aggregation implies $|E| \cdot n_I \cdot f_\phi$ complete computations of overlaps and joint probabilities. The worst case therefore is of complexity $O(n_F \cdot n_I)$.

4 Experimental Evaluation

In this section, we report on experiments evaluating the discussed operators. The goal of those experiments is the validation of our analytical results for the selected operators. Further, based on concrete results, we can quantify the costs for reading and writing data and the specific computations involved in operator processing steps.

4.1 Implementation and Setup

We extended an existing proprietary engine that computes complex analytical queries by means of so-called *Calculation Views* (CV). Those views enable OLAP analysis functionality as well as applying custom operations provided as Python or C++ implementations.

For a more detailed explanation of the underlying architecture and the concept of CVs, see [JLF10]. Further, for the computation of statistic functions (e.g., for copula construction at runtime) we rely on the statistic library IMSL³. Physically, the data are stored and accessed per column. Since our operators operate only on one or two columns at most of the time, this setting is beneficial for many operations.

Experimental Setup For the following experiments, we used a dual CPU workstation with 4GB of main memory running Windows Vista 64bit. The goal was to investigate the required run-times for selected operators. To validate the influence of factors identified in Section 3 above on actual run-times, we varied both the parameterization of operators and the scaling factors of the underlying TPC-H⁴ data as well as the characteristics of uncertain input data (represented mostly by histograms). Note that, in the general case, intermediate results derived by an operator are not persisted unless stated otherwise.

4.2 Experiments

Loading Histogram Data As a basis for most of the other operators, histogram data needs to be loaded from the database into our internal histogram structure. Figure 4 shows the costs induced by loading histograms with varying numbers β of bins. Note that for the current prototype, those costs are far from optimal due to the fact that we access the internal tables storing our histogram data via SQL statements rather than internal table searches. Clearly, load times increase linearly with the number of fetched histograms. However, repeated access to individual values or small sets of values causes relatively higher costs.

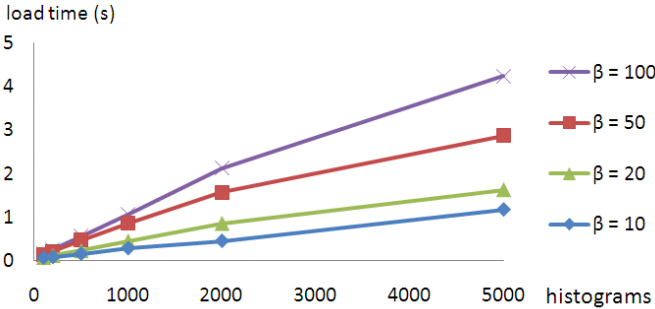


Figure 4: Times for loading histograms for a number of uncertain values

Derivation Subsequently, we use the `lineitem` table from the TPC-H benchmark as a basis for deriving distributions. We derive both histogram-based representations with vary-

³<http://www.vni.com/products/imsl/>

⁴<http://tpc.org>

ing numbers of bins and symbolic representations over the values of the `extendedprice` attribute. We assume that the considered data follows either one of a uniform, a Gaussian, or a Gamma distribution. Using scaling factors $s = 0.1$ and $s = 1.0$ results in $n_F = 600k$ and $n_F = 6M$ attribute values, respectively. Depending on an optional grouping, we derive a *total* distribution over all values, or $20k$ and $200k$ distributions for each `lineitem,partkey`, respectively. As shown in Table 1, run-times increase almost linearly with the size of n_F when we derive one distribution from all fact values. In the grouped derivation, we observe a slightly stronger increase in the case of histogram derivations, which we attribute to the high memory allocation costs. This factor also causes slightly rising run-times as β increases, even though the bin allocation as such is constant at $O(1)$. For the derivation of symbolic representations, run-times increase perfectly linear with n_F .

Table 1: Run-times (*sec*) for derivation of different distribution representations over a total of $n_F = 600k$ values ($s = 0.1$) and $n_F = 6M$ values ($s = 1.0$) of attribute `lineitem.extendedprice`.

Scale	Distribution Representation					
	(Equi-width) Histogram			Symbolic		
	$\beta = 10$	$\beta = 20$	$\beta = 100$	Uniform	Gaussian	Gamma
$s = 0.1, total$		0.165		0.156	0.28	0.28
$s = 0.1, grouped$	0.57	0.60	0.61	0.65	0.92	0.92
$s = 1.0, total$		1.62		1.55	2.82	2.82
$s = 1.0, grouped$	6.55	6.87	6.96	6.50	8.95	8.95

Table 2: Run-times (*sec*) for converting 1000 values from symbolic to histogram representations.

Source	Target Histogram Representation β			
	10	20	50	100
<i>Uniform</i>	0.136	0.19	0.3	0.556
<i>Gaussian</i>	0.115	0.173	0.338	0.619
<i>Gamma</i>	0.136	0.176	0.364	0.692

Table 3: Run-times (*sec*) for converting 1000 histogram-based representations to symbolic representations.

Source	Target Distribution		
	Uniform	Gaussian	Gamma
$\beta = 10$	0.025	0.06	0.062
$\beta = 50$	0.028	0.067	0.068
$\beta = 100$	0.035	0.080	0.082

Conversion Tables 2 and 3 show results for converting symbolic into histogram representations with varying β and for converting histograms into assumed symbolic representations, respectively. We can see in Table 2 that the observed costs are relatively higher for low values of β , which results from setup and loading costs. Beyond this initial cost, run-times increase almost linear with β , due to the fact that we must compute discrete density values by integration within each of the β bins as discussed in Section 3.2. Note that the concrete cost for potential further distribution functions will vary depending on the concrete implementation (e.g., through a call to an external library) of their integration. Table 3 shows run-times for deriving function parameters of assumed distribution functions from source histograms. Computation costs increase only slightly with the size

of β . The total costs are strongly dominated by loading times, increasing linearly with β as already shown in Figure 4.

Modification We evaluated both the modification of values based on an update value x_{new} provided a priori and applying value-based conditions. Different from the other operators, in the case of *MOD*, we write back the bin-wise delta from the old to the new value. Figure 5 shows the results of modifying histograms with varying β based on a threshold on the distribution support; that is, all bins with right bounds below a threshold were modified (e.g., set to 0). The threshold was varied so that the modified fraction f_m increased from 0.0 to 1.0. One can see that run-times increase linearly with f_m as only modified bins are written back. The increase of run-times is also linear in β . We observe a stronger increase for $\beta = 100$, which is due to current implementational restrictions of our prototype.

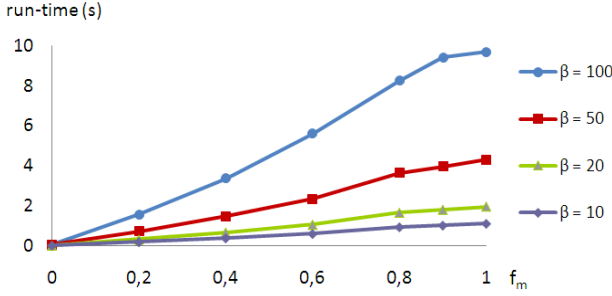


Figure 5: Run-times for modifying 100 histograms, applying (value-based or frequency-based) conditions affecting various fractions of the histograms

Correlation Introduction To evaluate run-times of the *COR* operator, we varied the characteristics of the copulas underlying the sampling- and ACR-based approaches as well as the distributions to be correlated. First, we evaluated the application of different copulas $C_{H,d}$ built from a bivariate Gaussian and T-student distribution H with correlation degrees $d = 0.4$ and $d = 0.8$, respectively. We further varied the number of samples (n_S) per copula $C_{H,d}$ and the number of bins (α) per ACR $\bar{C}_{H,d}^\alpha$. Both $C_{H,d}$ and $\bar{C}_{H,d}^\alpha$ were applied to two histograms \bar{P}_x^{20} and \bar{P}_y^{20} to yield a result histogram $\bar{P}_{x,y}^{20,20}$. Table 4 displays the run-times required for computing $\bar{P}_{x,y}^{20,20}$, averaged over 100 runs each. For the sampling-based approach, we must include the times for copula construction and for deriving $\bar{P}_{x,y}^{20,20}$. In contrast, for the ACR-based approach, we exclude copula construction times since we only need to access the precomputed ACR histograms and compute the quantiles for the $n_{S_{ACR}}$ artificially derived samples. In this case, we fix $n_{S_{ACR}} = 100k$. Table 4 shows that the run-times of the ACR-based cases are almost constant at about the time required for processing the respective copulas $C_{T(1),d}$ and $C_{G,d}$ using $n_S = 5k$ and $n_S = 10k$ samples, respectively. The constant behavior is due to the fact that we keep the number of

$n_{S_{ACR}}$ constant for all applied ACRs, irrespective of the number of α . The displayed run-times for the sampling-based approach increase linearly with the used number of samples n_S . As an indication – although the issue of accuracy is not further discussed in this paper – the result accuracies reached by using copulas of 20k and 40k samples are comparable to those achieved when using the respective ACR with $\alpha = 40$ bins.

Table 4: Run-times (in *ms*) for deriving $\bar{P}_{x,y}^{20,20}$ through sampling approach and ACR processing

Copula	n_S			ACR-based (α)		
	5k	10k	40k	10	20	40
$C_{Gauss,0.4}, C_{Gauss,0.8}$	123	157	361	150	150	150
$C_{T(1),0.4}, C_{T(1),0.8}$	140	189	474	150	150	150

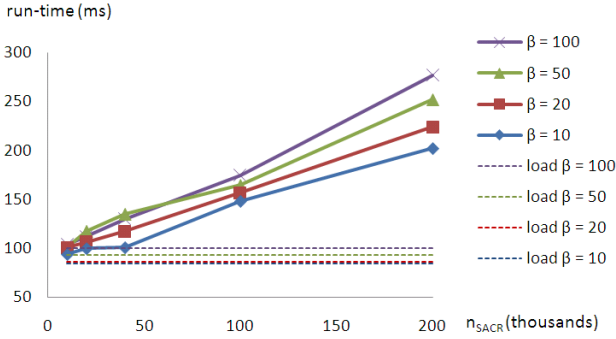


Figure 6: Run-times for ACR-based correlation introduction with varying parameters

We further varied the number of β_x and β_y used to represent P_x and P_y as well as the number of $n_{S_{ACR}}$. The resulting run-times are shown in Figure 6. We can see that there is an initial cost for setting up the operator and loading the data, which clearly dominates run-times especially for small β and $n_{S_{ACR}}$. Beyond this initial cost factor, we see an increase linear in $n_{S_{ACR}}$ due to the cost for each additional sample inversion (quantile computation). With increasing numbers of β_x and β_y (simultaneously set to 10, 20, 50, or 100 bins, respectively) we can see increasing run-times slightly below the assumed logarithmic increase due to the increased cost of each quantile computation, as discussed in Section 3.4.

Temporal Aggregation The efficiency of AGG^T is subject to many variations as described above. We now evaluate the influence of β_{t_i} and β_{d_i} , as well as the fraction f_ϕ of the potential occurrence intervals I_{ipq} overlapping T . We apply $SUM^{[10,15]}$ for 1000 artificial events $e_i \in E$. Start times t_i and durations d_i are uniformly distributed over $[0, 5]$, each represented by a corresponding histogram with $\beta_{t_i} = \beta_{d_i} = 5$. The results are displayed in Figures 7(a) and 7(b).

We investigate the variation of β by aggregating over a number of 1000 events associated with varying β_{t_i} and β_{d_i} , respectively. The portion f_ϕ of overlapping occurrence intervals

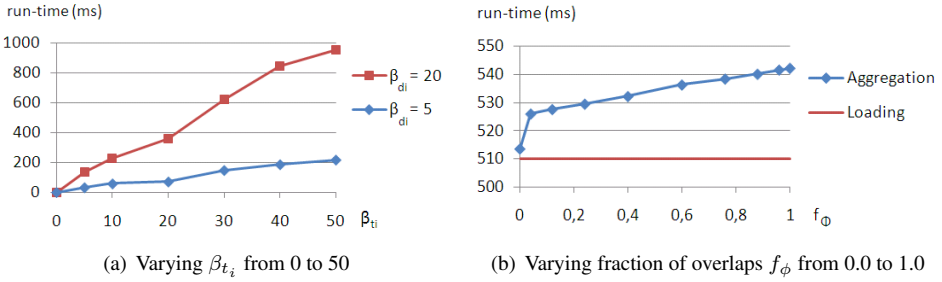


Figure 7: Temporal aggregation over 1k events with varying start time and duration characteristics

is kept stable (at 100%) by ensuring that, for every variation, $h_{t_i} < l_{\underline{T}} \wedge l_{t_i} + l_{d_i} > l_{\underline{T}}$. The results are shown in Figure 7(a). Results for varied β_{d_i} are similar given the portion of potential overlaps is similarly kept stable at 100%. To vary the portion f_ϕ of overlapping I_{ipq} from 0.0 to 1.0, we keep t_i and \underline{T} constant and calculate $AGG^{\underline{T}}$ for $l_{d_i} = 0, \dots, 10$ and $h_{d_i} = l_{d_i} + 5$. The resulting run-times are shown in Figure 7(b). In both experiments, the observed behavior is in line with the results of Section 3.5, reflecting a linear rising in run-times for increased β_{t_i} , β_{d_i} , and f_ϕ , respectively. An exception occurs for $f_\phi = 1/25$, where we observe an initially stronger increase. This is due to the fact that for $f_\phi = 0.0$ all events lie outside \underline{T} and do not induce any costs for testing of overlapping intervals, while for any $f_\phi > 0.0$ those tests imply an initial cost.

5 Efficiency and Optimization of the Analysis Process

So far, we discussed the efficiency of individual operators with respect to their specific characteristics. We now address selected issues of optimization in the face of large amounts of input data and the application of composed operator sequences in order to enable lower response times for their interactive and iterative application in analysis processes.

An example process In Figure 8 we illustrate the subsequent application of selected operators implementing UC 2. Recall that we want to analyze costs associated with a set of indeterminate deployments E_{dep} during a specified time interval, where the deployment of a part follows its uncertain delivery. Consider as the basis for our analysis a data warehouse storing information about line items and associated orders as represented in the `lineitem` and `order` tables. We want to prospect the probable deployment costs for a group of line items during the next weeks. We assume that their times to delivery (`ttd`) (computed from the `order.orderdate` and `lineitem.receiptdate` attributes) will behave similar to the distribution of delivery times observed in the historic data. To reflect this assumption, for each `lineitem.partkey`, we derive a histogram (EWH) \bar{P}_{t_i} over `ttd` for all delivered items. We view \bar{P}_{t_i} as the distribution of the start time of a prospective deployment event e_i for an item ordered today (viewing "today" as day 0). For

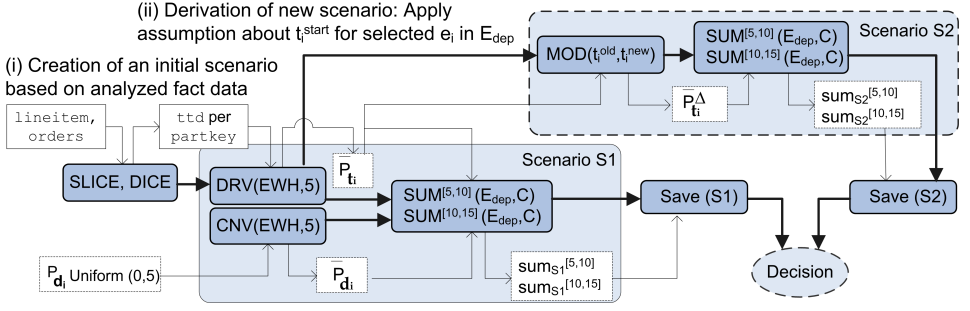


Figure 8: Process illustrating UC 2 and the incremental derivation of scenarios

simplicity, we assume a constant cost $c_i = 100$ and a duration d_i following a uniform distribution in $[0, 5]$ (converted to \bar{P}_{d_i}) for all deployment events $e_i \in E_{dep}$. The aim of our analysis is (i) to compute prospective deployment costs induced by selected orders during time intervals $[5, 10]$ and $[10, 15]$ and (ii) to investigate, in the case of an unfavorable cost situation, alternative delivery scenarios. The first step is achieved through an application of $SUM^{[5,10]}(E_{dep}, C)$ and $SUM^{[10,15]}(E_{dep}, C)$. The latter involves the modification of deployment start times t_i based on the user's assumption followed by a second aggregation over E_{dep} , now including the modified temporal information.

5.1 Iterative creation and computation of scenarios

In our exemplary use case UC 2, the user wants to analyze the influence of applying express delivery on the deployment costs induced during a considered time frame. To this end, as shown in Figure 8, he modifies the start times t_i of a selected subset of E_{dep} , creating a new scenario $S2$. He then needs to analyze the modified data underlying $S2$, e.g., by repeating the described aggregation over intervals $[5, 10]$ and $[10, 15]$. The results scenarios can be compared, stored, or further processed. Naturally, we want to reuse as many results as possible throughout this process. To this end, we must provide information about their derivation and evaluate it in the processing of operators. It is important to note that most of the derived results are kept in memory as intermediate results, enabling fast access and iterative application of different operators. Of course, we can persist results to enable their reuse at a later point in time.

Incorporating modifications and insertions The creation of a new scenario virtually always goes along with modifications to some minor part of the underlying data. For example, the scenario described above is derived on the assumption of modified delivery times for a group of items. Note that we can calculate the succeeding aggregation very efficiently given the fraction of modified times is relatively small. In particular, rather than applying AGG^T to all $e_i \in E_{dep}$, we only need to compute SUM^T over the affected events, using the delta values t_i^Δ that represent the previous modification. We can then

compute sum_{S_2} as $sum_{S_2} = sum_{S_1} + sum_{\Delta}$. Modifications can be incorporated in the described iterative fashion only if we can preserve the semantics of the applied operators. For example, we can apply a similar step to update previously derived histograms or distribution parameters to incorporate modified or new fact data in an iterative fashion, rather than recomputing the complete distribution. Similarly, we can update a bivariate distribution (derived using *COR*) when one of the marginals is modified. This is because internally, *COR* essentially relies on summing up the joint densities in the result histogram. Thus, the same iterative approach as above can be applied. Note that we can not use this approach when the analysis process includes operators whose semantics are not preserved under modifications, e.g., for the computation of extrema (both in the sense of standard aggregation and the computation of MIN^T and MAX^T). A comprehensive consideration of how new data and modifications can be incorporated in the execution of (sequences of) operators is yet outstanding.

5.2 Parallelization

Besides optimizing the calculation of succeeding operators based on the provenance of intermediate results, we also need to address the issue of long response times due to large amounts of data processed by operators such as *DRV*. To this end, we considered different forms of parallelization. For many of our operators, a large part of their overall costs is determined by loading and processing individual columns. Those can be executed independently, returning relatively small results which are merged in a final step. In previous work, we applied alternative ways of parallel loading and processing of underlying input data. We evaluated parallelization of computations executed within an operator and parallel processing of operators between cores in [ERM⁺10], where we exemplified our approaches using the data-intensive operator *DRV*. The reported results show that in cases of large amounts of input data, parallel loading and processing of partitioned data over many cores is beneficial due to dominant loading times. Conversely, when operators process relatively small amounts of data, we can apply threaded execution within a single operator.

6 Related Work

Existing approaches for uncertain data management foremost focus on areas such as the management of sensor data, information extraction results, or scientific data. In this context, those approaches mainly address the representation, indexing, and analysis of data represented through tuple alternatives [HAKO09, SD07, ABS⁺06] and values distributed over discrete or continuous domains [SMM⁺08, AW09]. The generally high complexity of queries over uncertain data is a well-known problem and has been discussed – among other issues – with respect to join evaluation [Che06], range predicates [DS07, CXP⁺04], and exact and approximate aggregate computation [MIW07].

The abovementioned aspects serve as valuable building blocks for the analysis part of the planning processes we envision. However, our work focuses on the specific aspects of derivation and modification of uncertainty and interdependencies in data. To incorporate those aspects, we apply symbolic and equi-width histogram representations of distribution functions. The use of histograms and the performance of different partitioning schemes, such as equi-depth or MaxDiff, have been investigated in depth (see, e.g., [PHIS96]) with respect to both their construction efficiency and accuracy. Likewise, the usage of histogram-based and symbolic representations for uncertain data management has been previously discussed, e.g., in [SMM⁺08, AW09]. We similarly exploit the histogram model to represent arbitrary distributions generically; in addition, our data model employs uni- and multivariate histograms to represent and efficiently handle modifications (deltas) and correlation information. The aspect of correlations in data was addressed previously primarily for the case of tuple alternatives and discrete value distributions. The approach reported in [SD07] uses graphical models to represent such dependencies in a factored fashion and discusses efficient inference-based query evaluation over the graphs. Although, in general, the graphical representation can be applied in the face of continuous distributions, [SD07] does not address the computation or introduction of correlation information by users. While the authors in [KO08] discuss efficient approaches for the introduction of "conditioning" constraints (e.g., implications and mutual exclusion) and queries over conditioned data, they similarly do not discuss the separate representation and introduction of *arbitrary* correlation to data as we do. Our previous work [ER10, ERM⁺10] introduced the general ideas of our support for scenario-based planning, but lacked a comprehensive discussion and evaluation of our operators' efficiency, foremost as regards the *COR* and *AGG^T* operators. In this paper, we addressed this open issue as a basic prerequisite to judge their practical applicability. The challenge of efficiently incorporating modified data in the face of scenario creation relates to the topic of data lineage. Lineage handling has been previously discussed in the context of probabilistic data, e.g., in [ABS⁺06, STW08] and in the broader context of view maintenance in data warehouses [CWW00] and data-centric workflows. Its application for optimizing scenario-based planning process constitutes an interesting new facet complementing previous research.

7 Conclusion and Future Work

In this paper, we extended our previous work on operators for derivation, analysis and modification of uncertain data in the context of scenario-based planning processes. We derived and discussed the complexity of these operators and created a basis for assessing them in different application scenarios. We also validated the analytical results through an experimental evaluation. Generally, we observe a dominating cost factor for loading histogram structures from the database, while the computation routines themselves are highly efficient and introduce only small additional costs with growing data complexity. We further highlighted opportunities for optimization concerning both parallelization and the incremental execution of steps in an analysis process, including the efficient derivation of related scenarios. Finally, we addressed related research topics touching on various

aspects of both the functional and performance-related aspects of the presented work.

Apart from enabling provenance handling in the scenario derivation process, another highly important factor of future work is an assessment of the accuracy of results derived in this process. Naturally, we cannot quantify the “correctness” of a computed result scenario since the future fulfillment of the applied assumptions is unknown. Still, we can measure discretization errors introduced through operator applications and quantify the resulting trade-offs between accuracy and efficiency. In this respect, both varying aspects of the data model and the applied operators can help enable manual and automatic optimization based on users’ preferences. For example, applying an alternative histogram partitioning scheme such as equi-depth could decrease approximation errors at the cost of lower construction and update efficiency. Conversely, a user might resort to approximate operators for the benefit of lower run-times. In this context, a complementary track of our work investigates approximate temporal aggregation based on a clustering of events with similar temporal associations. A comprehensive investigation of the exemplified trade-offs is subject to future work.

References

- [ABS⁺06] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A System for Data, Uncertainty, and Lineage. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1151–1154. VLDB Endowment, 2006.
- [AW09] Parag Agrawal and Jennifer Widom. Continuous Uncertainty in Trio. In *MUD*. Stanford InfoLab, 2009.
- [Che06] Reynold Cheng. Efficient join processing over uncertain data. In *In Proceedings of CIKM*, pages 738–747, 2006.
- [CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.
- [CXP⁺04] Reynold Cheng, Yuni Xia, Sunil Prabhakar, Rahul Shah, and Jeffrey Scott Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 876–887. VLDB Endowment, 2004.
- [DS07] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.
- [ER10] Katrin Eisenreich and Philipp Rösch. Handling Uncertainty and Correlation in Decision Support. In *Proceedings of 4th Workshop on Management of Uncertain Data at VLDB 2010*, September 2010.
- [ERM⁺10] Katrin Eisenreich, Philipp Rösch, Volker Markl, Gregor Hackenbroich, and Robert Schulze. Handling of Uncertainty and Temporal Indeterminacy for What-if Analysis. In *Proceedings of Workshop on Enabling Real-Time Business Intelligence at VLDB 2010*, September 2010.

- [HAKO09] Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. MayBMS: A Probabilistic Database Management System. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 1071–1074, New York, NY, USA, 2009. ACM.
- [JLF10] Bernhard Jäcksch, Wolfgang Lehner, and Franz Faerber. A Plan for OLAP. In *EDBT*, pages 681–686, 2010.
- [JXW⁺08] Ravi Jampani, Fei Xu, Mingxi Wu, Luis L. Perez, Christopher Jermaine, and Peter J. Haas. MCDB: A Monte Carlo Approach to Managing Uncertain Data. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 687–700, New York, NY, USA, 2008. ACM.
- [KO08] Christoph Koch and Dan Olteanu. Conditioning probabilistic databases. *Proc. VLDB Endow.*, 1(1):313–325, 2008.
- [MIW07] Raghotham Murthy, Robert Ikeda, and Jennifer Widom. Making Aggregation Work in Uncertain and Probabilistic Databases. Technical Report 2007-7, Stanford InfoLab, June 2007.
- [MST07] G. Mayor, J. Suner, and J. Torrens. Sklar’s Theorem in Finite Settings. *Fuzzy Systems, IEEE Transactions on*, 15(3):410–416, june 2007.
- [Nel06] Roger B. Nelsen. *An Introduction to Copulas*. Springer Series in Statistics. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
- [PHIS96] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. *SIGMOD Rec.*, 25(2):294–305, 1996.
- [SD07] Prithviraj Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 596–605, 2007.
- [Skl59] A. Sklar. Fonctions de repartition à n dimensions et leurs marges. *Publications de l’Institut de Statistique de L’Universite de Paris*, 8:229–231, 1959.
- [SMM⁺08] Sarvejeet Singh, Chris Mayfield, Sagar Mittal, Sunil Prabhakar, Susanne Hambrusch, and Rahul Shah. Orion 2.0: Native Support for Uncertain Data. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD*, pages 1239–1242, New York, NY, USA, 2008. ACM.
- [STW08] Anish Das Sarma, Martin Theobald, and Jennifer Widom. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1023–1032, Washington, DC, USA, 2008. IEEE Computer Society.

Resolving Temporal Conflicts in Inconsistent RDF Knowledge Bases

Maximilian Dylla* Mauro Sozio Martin Theobald
{mdylla, msozio, mtb}@mpi-inf.mpg.de
Max-Planck Institute for Informatics (MPI-INF)
Saarbrücken, Germany

Abstract: Recent trends in information extraction have allowed us to not only extract large semantic knowledge bases from structured or loosely structured Web sources, but to also extract additional annotations along with the RDF facts these knowledge bases contain. Among the most important types of annotations are spatial and temporal annotations. In particular the latter temporal annotations help us to reflect that a majority of facts is not static but highly ephemeral in the real world, i.e., facts are valid for only a limited amount of time, or multiple facts stand in temporal dependencies with each other. In this paper, we present a *declarative reasoning framework* to express and process temporal consistency constraints and queries via first-order logical predicates. We define a subclass of first-order constraints with temporal predicates for which the knowledge base is guaranteed to be satisfiable. Moreover, we devise efficient grounding and approximation algorithms for this class of first order constraints, which can be solved within our framework. Specifically, we reduce the problem of finding a consistent subset of time-annotated facts to a scheduling problem and give an approximation algorithm for it. Experiments over a large temporal knowledge base (T-YAGO) demonstrate the scalability and excellent approximation performance of our framework.

1 Introduction

Despite the great advances of Web-based information extraction (IE) techniques in recent years, the resulting knowledge bases still face a significant amount of noisy and even inconsistent facts. These knowledge bases are typically captured as RDF facts, with some of the most prominent representatives being DBpedia, FreeBase, and YAGO. The very nature of the largely automated extraction techniques that these projects employ however entails that the resulting RDF knowledge bases may face a significant amount of incorrect, incomplete, or even inconsistent factual knowledge (which is often summarized under the term *uncertain data*). A knowledge base becomes inconsistent only through the presence of additional *consistency constraints*, which are typically provided by a human knowledge engineer according to some real-world-based domain model. In general, we call a knowledge base *inconsistent* if not all these provided consistency constraints are satisfied with

*The author has partially been supported by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

respect to the facts captured by the knowledge base. Resolving these inconsistencies thus requires some form of *consistency reasoning*, for example, by selecting a consistent subset of the facts contained in the knowledge base, and by considering only this subset for answering queries.

By default, we assume facts in the knowledge base to be *true*, and (implicitly) all facts not contained in the knowledge base to be *false*, an approach generally known as *closed-world assumption*. Consistency constraints may however put two or more facts in the knowledge base into conflict with each other, thus rendering the knowledge base inconsistent (i.e., *unsatisfiable*) under the assumption that all facts contained in it are *true*. For example, an extractor might erroneously extract two different birth places of David Beckham, expressed as the two RDF facts *bornIn(David_Beckham, Leytonstone)* and *bornIn(David_Beckham, Old_Trafford)* in our knowledge base. Without an explicit constraint, which puts these two facts into conflict with each other, there is no formal inconsistency in a knowledge base containing these two facts. Therefore, queries asking for the birth place of David Beckham would return both answers. With an explicit (first-order) logical consistency constraint of the form

$$\forall x, y, z \text{ bornIn}(x, y) \wedge \text{bornIn}(x, z) \rightarrow y = z$$

however, we can express that only one of the two above facts may be true in the real world. Hence, the reasoner (ideally at query-time) could decide which of the two facts to return as answer. Moreover, multiple of these constraints may overlap, such that the truth value of a fact may depend on multiple constraints. In turn, the constraints may put multiple, partially overlapping (sub-)sets of facts contained in the knowledge base into conflict with each other. Generally, Boolean reasoning within this family of SAT problems is NP-hard, and for general first-order formulas the constraints may not be satisfiable at all. In other words, there may exist no truth assignment to facts (even regardless of the actual facts) in the knowledge base such that all constraints are satisfied.

Temporal annotations add another dimension of complexity to reasoning with RDF facts. With temporal annotations, we can not only express general constraints among facts but also add a finer granularity to the consistency reasoning itself. Only with time information, we can, for example, express that a person should only be *married to* at most one other person at a time, that a soccer player can *play for* only one club at a time, or that a person had to be *married to* another person before they got *divorced*, and so on. Even when using simple time intervals for the representation of temporal annotations with such disjointness and precedence constraints, the satisfiability problem is known to be NP-hard [GS93].

Thus, our goal in this work is to identify a canonical set of first-order constraints, for which we know that they are satisfiable over a given knowledge base, and to provide an efficient framework for resolving temporal conflicts directly at query-time.

1.1 Contributions

The contributions of the work presented in this paper are three-fold:

- **Declarative reasoning framework for consistency constraints and queries.** We focus on temporal consistency reasoning over large, uncertain, and potentially incon-

sistent knowledge bases. Our constraints are expressed as first-order logical Horn formulas with temporal predicates, a setting which leaves the satisfiability problem NP-hard¹, and which may result in unsatisfiable constraints. We thus define a subclass of Horn constraints with temporal predicates whose satisfiability is guaranteed, and which we can solve efficiently in terms of both grounding the first-order formulas and resolving conflicts among the grounded facts (Section 3.1). Both constraints and queries can be specified by the user in a fully declarative way.

- **Efficient Approximation Algorithm.** We develop a linear-time algorithm for checking whether a general set of first-order constraints is included in our previously defined solvable subclass of constraints (Section 3.1). Moreover, we introduce a grounding procedure whose running time linearly depends both on the constraints and the number of query-matches contained in the knowledge base (Section 3.2). Finally, we present a procedure for efficiently and effectively resolving temporal conflicts among facts contained in the knowledge base (Section 3.2), which remains an NP-hard problem also for our class of constraints, and for which we devise an efficient approximation algorithm (based on results from event scheduling) for solving these conflicts.
- **System and Experiments.** We experimentally evaluate our system over the T-YAGO [WZQ⁺10] knowledge base, consisting of 270,000 temporal facts, and handcrafted consistency constraints (Section 4). Our evaluation shows that the system scales very well and at the same time features excellent performance in terms of approximation quality.

The remainder of this paper is organized as follows. In Section 2, we provide a formal definition of our data model and the first-order constraints. In Section 3, we define the subclass of constraints we tackle, and we discuss offline and online computations required to solve these constraints over a set of given base facts (the knowledge base). Our experimental results are shown in Section 4. Continuing with related work in Section 5, we conclude our work in Section 6.

2 Data Model, Constraints, and Problem Statement

2.1 Data and Representation Model

Uncertain Temporal Knowledge Base. We define a *knowledge base* $\mathcal{KB} = \langle \mathcal{F}, \mathcal{C} \rangle$ as a pair consisting of a set of (weighted and temporal) facts \mathcal{F} and a set of first-order (temporal) consistency constraints \mathcal{C} (the latter are discussed in Section 2.2). To encode facts, we employ the widely used Resource Description Format (RDF), in which facts $\mathcal{F} \subseteq \text{Rel} \times \text{Entities} \times \text{Entities}$ are stored as *triples* consisting of a relation and a pair of entities. Moreover, we extend the original RDF triplet structure in two ways: first, to express *uncertainty* about a fact’s correctness, we associate a positive, real-valued *confidence weight* $w(f)$ with each fact $f \in \mathcal{F}$ (denoted by the function $w : \mathcal{F} \rightarrow \mathbb{R}^+$); and second, to include time information into our knowledge base, we also assign a *time interval* of the form $[t_b, t_e)$ to each fact f . The weights $w(f)$ can be interpreted as the confidence for the

¹The satisfiability problem of propositional Horn-SAT is in \mathcal{P} , whereas first-order Horn-SAT (with variables being all-quantified) is NP-hard.

fact being *true*, where a higher value denotes a higher confidence, while the time interval $[t_b, t_e]$ specifies the begin time t_b and end time t_e during which the fact may be valid, i.e., during which it may be *true*. Outside their validity intervals, facts are assumed to be *false*. Time intervals, as well as temporal predicates for logical reasoning with these intervals, are defined more formally in the next subsection.

Time Intervals and Temporal Predicates. In our setting, the set of *time intervals* $\mathcal{T} \subseteq \mathbb{N}_0 \times \mathbb{N}_0$ is composed of all possible (half-open) time intervals of the kind $[t_b, t_e)$ with $t_b < t_e$. For presentation purposes, we will denote intervals as if they range over years, like the interval $[1990, 2010)$ which starts in 1990 and ends in 2009. Our reasoning framework however supports arbitrary continuous intervals over real numbers.

The set of relations is $Rel = Rel_E \dot{\cup} Rel_A$ is split into a set of *extensional relations* Rel_E (like, e.g., *bornIn* or *graduatedFrom*), which are captured purely by facts stored in the knowledge base, and a set of *arithmetic relations* Rel_A (e.g., *equal* “=”, or *notEqual* “ \neq ”), which are evaluated by the reasoner “on demand” based on their arguments (i.e., all their arguments become constants when the formulas are grounded).

In addition to the common arithmetic predicates for expressing the equality and inequality of two arguments, we deploy *temporal predicates* $Rel_T \subseteq Rel_A$ as a subset of the arithmetic predicates we consider in our reasoning framework. Temporal predicates enable us to reason about the temporal relationships among facts based on their time intervals. For example, we say that two time intervals *overlap* if they share a common time interval; otherwise they are *disjoint*. Further, a time interval $[t_{b_1}, t_{e_1})$ is *before* another interval $[t_{b_2}, t_{e_2})$ if $t_{e_1} \leq t_{b_2}$, which also implies that they are disjoint (see, for example, seminal work by Allen et al. [All83] for an overview of temporal relations among intervals).

Example 1. Besides the first line expressing that David Beckham was born in Leytonstone in 1975 with weight 9.0, Figure 1 contains four additional facts related to him.

```

fbornBL := bornIn(David_Beckham, Leytonstone, [1975, 1976))9.0
fbornBOT := bornIn(David_Beckham, Old_Trafford, [1999, 2000))2.0
fplaysBMU := playsForClub(David_Beckham, Manchester_United, [1993, 2004))8.0
fplaysBB := playsForClub(David_Beckham, 1.FC_Barcelona, [1999, 2001))6.0
fplaysBE := playsForNational(David_Beckham, England_National_Team, [1992, 2011))1.0

```

Figure 1: The content of \mathcal{F} in our running example.

2.2 Constraints and Queries

Consistency Constraints. A *consistency constraint* in our reasoning framework is a first-order logical Horn formula with exactly two extensional predicates $rel_{E_1}, rel_{E_2} \in Rel_E$, an optional arithmetic (but non-temporal) predicate $rel_A \in Rel_A \setminus Rel_T$ in the body, and exactly one temporal predicate $rel_T \in Rel_T \cup \{false\}$ as head literal. Constraint (1) denotes the general template of consistency constraints we consider in the following.

$$rel_{E_1}(e_1, e_2, t_1) \wedge rel_{E_2}(e_1, e_3, t_2) \wedge rel_A(e_2, e_3) \rightarrow rel_T(t_1, t_2) \quad (1)$$

All occurring variables, where e_1, e_2, e_3 represent entities and t_1, t_2 stand for time intervals, are implicitly universally quantified. We require rel_{E_1} and rel_{E_2} to share e_1 as their first argument, and the optional arithmetic predicate rel_A must hold the remaining variables e_2 and e_3 as its arguments.

Queries. As opposed to constraints, queries are conjunctions of extensional predicates, where all variables are implicitly existentially quantified. For example, the query

$$playsForClub(David_Beckham, club) \quad (2)$$

may be imposed by a user to ask: “Which clubs did David Beckham play for?”

2.3 Reasoning Framework and Semantics

When we instantiate (i.e., *ground*) the literals in the first-order consistency constraints \mathcal{C} and replace them by facts, we obtain propositional formulas. Then the facts represent propositional literals, which can be either set to *true* or *false* by the reasoner. Arithmetic predicates with constants are immutable in a propositional sense, i.e., they are always either *true* or *false*, depending on the constants and the semantics of the predicate. For example, the two entities *Beckham* and *Ronaldo* are never *equal* under the Unique Name Assumption of the underlying RDF data model, and the two time intervals [1999, 2003) and [2004, 2006) can never *overlap*. Thus, in each grounded instance of a constraint, only the two literals with extensional predicates become actual Boolean variables and can be assigned a truth value by the reasoner. According to the structure of the constraints described above, two facts are in conflict with each other if they are contained in a propositional instance of a constraint whose (temporal) head literal is *false*, which implies that the entire constraint evaluates to *false* given that both facts are *true*. Hence, in order to resolve such an inconsistency, we have to set at least one of the extensional facts to *false*.

2.4 Constraint Types

Depending on the choice of the constraints, the combinatorial complexity of resolving conflicts is varying, making it crucial to decide which constraints we allow to be formulated. In the following, we consider three kinds of constraints, which handle a significant number of possible scenarios:

- Temporal disjointness
- Temporal precedence
- Mutual exclusion

Disjointness. To express that the intervals of any two facts from the same extensional relation rel_E (e.g., *playsForClub*) are non-overlapping, we utilize the following template to express disjointness constraints.

$$rel_E(e_1, e_2, t_1) \wedge rel_E(e_1, e_3, t_2) \wedge e_2 \neq e_3 \rightarrow disjoint(t_1, t_2) \quad (3)$$

Example 2. We express that a player can only play for one club at a time by replacing rel_E in (3) by *playsForClub*:

$$playsForClub(e_1, e_2, t_1) \wedge playsForClub(e_1, e_3, t_2) \wedge e_2 \neq e_3 \rightarrow disjoint(t_1, t_2) \quad (4)$$

The facts f_{playsBMU} , f_{playsBB} are in conflict with respect to (4), as their time intervals [1993, 2004), [1999, 2001) share a time interval, which makes them non-disjoint.

Precedence. Restricting that the time interval of an instance of rel_{E_1} ends before the interval of a fact with rel_{E_2} starts is reflected by the following template for precedence constraints.

$$rel_{E_1}(e_1, e_2, t_1) \wedge rel_{E_2}(e_1, e_3, t_2) \rightarrow \text{before}(t_1, t_2) \quad (5)$$

We note that in both other constraints (see Equations (3) and (7)), there is only one extensional relation. Here there are two, namely rel_{E_1} and rel_{E_2} .

Example 3. A very natural constraint in the sports domain is that the birth date of a person should precede the participation in a sports club.

$$\text{bornIn}(e_1, e_2, t_1) \wedge \text{playsForClub}(e_1, e_3, t_2) \rightarrow \text{before}(t_1, t_2) \quad (6)$$

Now, neither f_{playsBMU} nor f_{playsBB} are in conflict with f_{bornBL} with respect to the constraint in (6), because [1975, 1976) ends before both [1993, 2004) and [1999, 2001) start. The situation is different for f_{bornBOT} , having the interval [1999, 2000) and hence being in conflict with f_{playsBMU} , f_{playsBB} under our precedence constraint (6).

Mutual Exclusion. Mutual exclusion, as the last type of constraints we consider, defines a set of facts which are all in conflict with each other, regardless of time. In general, a relation rel_E with a differing argument must not occur as expressed by the template:

$$rel_E(e_1, e_2, t_1) \wedge rel_E(e_1, e_3, t_2) \wedge e_2 \neq e_3 \rightarrow \text{false} \quad (7)$$

Example 4. Another very natural constraint in the domain of people is that a person cannot be born in multiple places.

$$\text{bornIn}(e_1, e_2, t_1) \wedge \text{bornIn}(e_1, e_3, t_2) \wedge e_2 \neq e_3 \rightarrow \text{false} \quad (8)$$

In our example, the two facts f_{bornBL} and f_{bornBOT} are in conflict with respect to (8).

2.5 Problem Statement

Assumptions. Our approach is based on two assumptions. First, the cardinality of \mathcal{F} can be huge. Second, the knowledge base may be evolving as new facts are extracted, i.e., the set of facts \mathcal{F} might be updated as the extraction process proceeds, or the constraints \mathcal{C} might be changing if we learn new relation types. Thus, enforcing consistency of the entire knowledge base might be both very expensive and abrasive with respect to changing constraints, which we aim to avoid by resolving conflicts between facts *dynamically* at query-time.

Problem Definition. Given a knowledge base $\mathcal{KB} = \langle \mathcal{F}, \mathcal{C} \rangle$, with weighted temporal facts \mathcal{F} , temporal consistency constraints \mathcal{C} and a query Q , we define $\mathcal{F}_Q \subseteq \mathcal{F}$ as the closure of all facts which are in conflict to a fact that matches Q .

Next, our goal is to resolve the conflicts by selecting a consistent subset of facts $\mathcal{F}_{Q, \mathcal{C}} \subseteq \mathcal{F}_Q$. In general, there may be several consistent subsets with the same cardinality, so

we extend our search by requiring that the sum of the weights of the consistent facts is maximized, as it is expressed by the following optimization problem:

$$\max_{\mathcal{F}_{Q,C} \subseteq \mathcal{F}_Q} \sum_{f \in \mathcal{F}_{Q,C}} w(f)$$

with the constraints:

$$\forall C \in \mathcal{C}. \text{Eval}(C, \mathcal{F}_{Q,C}) \equiv \text{true}$$

Here, *Eval* is the logical evaluation of all instances of the formula C by setting all facts in $\mathcal{F}_{Q,C}$ to *true* and all facts in $\mathcal{F}_Q \setminus \mathcal{F}_{Q,C}$ to *false*.

Finally, we return the matches to Q within $\mathcal{F}_{Q,C}$ as answers to the query.

Hardness. We show that the above problem contains the NP-hard Maximum Weight Independent Set problem.

Imagine a general graph. We introduce one relation for each vertex and one precedence constraint (5) for each edge, such that the constraint holds exactly the corresponding two relations which are connected by the edge. Finally, we create one fact for each relation while using always the same arguments, the same time-interval, and the weight of the corresponding vertex. It follows that a solution to the above problem is a solution to the Maximum Weight Independent Set problem, which is NP-hard.

3 Algorithm

The core of our framework is a scheduling algorithm which we employ to resolve conflicts between facts. In short, scheduling problems enclose a number of scheduling jobs which should be assigned to time slots on a number of scheduling machines, such that the machines do not exceed their capacities. In this section, we develop an algorithm which maps each fact to a scheduling job and consistency constraints to scheduling machines, such that a maximum-weight feasible schedule corresponds to a maximum-weight subset of conflict-free facts. This section is structured in accordance to the general flow of our framework as described in Algorithm 1. There are two phases, where the former deals with precomputations (Section 3.1, corresponding to Lines 1–4) and the latter (Section 3.2, corresponding to Lines 6–12) with computations at query-time.

As a first step, in Line 1 we translate the constraints \mathcal{C} to an equivalent, more compact representation as a *constraint graph* $G_{\mathcal{C}}$ (Section 3.1.1), where vertices and edges correspond to extensional relations and corresponding constraints, respectively. In Line 4, we cover the constraint graph with a number of subgraphs called *machine graphs* \mathcal{G}_M (Section 3.1.2). Each of the machine graphs represents a scheduling machine. Beforehand, Algorithm 1 checks in Lines 2 and 3, whether such a covering with machine graphs (scheduling machines) is possible and otherwise rejects the constraints.

Turning to the computations at query-time, in Line 6 (and more detailed in Section 3.2.1) the constraint graph is leveraged to obtain the set of facts \mathcal{F}_Q comprising the matches to the query together with their closure of conflicting facts. Then we strive to obtain the consistent subset $\mathcal{F}_{Q,C} \subseteq \mathcal{F}_Q$ in Line 12 to display the answer. Thereby, we exploit that the

extensional predicates in a constraint share a variable (see Section 2.2), which enables us to resolve the conflicts separately for each entity $e \in FirstArg = \{e \mid rel_E(e, e_2, t) \in \mathcal{F}_Q\}$ which instantiates this variable. Hence, $\mathcal{F}_{Q,e} = \{f \mid f \in \mathcal{F}_Q, f = rel_E(e, e_2, t)\}$ denotes the set facts, which are relevant to the query and which contain the entity e as their first argument. In Line 10, we invoke the actual scheduling algorithm (Section 3.2.2) for each of the subsets $\mathcal{F}_{Q,e}$ passing the machine graphs (scheduling machines) \mathcal{G}_M as an additional argument. It finally returns the set of query-relevant, consistent facts $\mathcal{F}_{Q,C,e}$ with respect to the entity e . The union of all sets $\mathcal{F}_{Q,C,e}$ forms $\mathcal{F}_{Q,C}$, which is the set of consistent facts which are relevant to the query Q .

Algorithm 1 Framework

Require: A knowledge base $\langle \mathcal{F}, \mathcal{C} \rangle$

Require: A set of queries \mathcal{Q}

- 1: Construct G_C from \mathcal{C} ▷ Section 3.1.1
 - 2: **if** G_C is not solvable **then**
 - 3: **return** error
 - 4: Construct the set of machine graphs \mathcal{G}_M from G_C ▷ Section 3.1.2
 - 5: **for all** $Q \in \mathcal{Q}$ **do**
 - 6: Ground Q to obtain the set $\mathcal{F}_Q \subseteq \mathcal{F}$ of relevant facts for Q ▷ Section 3.2.1
 - 7: $\mathcal{F}_{Q,C} := \emptyset$
 - 8: **for all** $e \in FirstArg := \{e \mid rel_E(e, e_2, t) \in \mathcal{F}_Q\}$ **do**
 - 9: $\mathcal{F}_{Q,e} := \{f \mid f \in \mathcal{F}_Q, f = rel_E(e, e_2, t)\}$
 - 10: $\mathcal{F}_{Q,C,e} := \text{RESOLVECONFLICTS}(\mathcal{F}_{Q,e}, \mathcal{G}_M)$ ▷ Algorithm 2, Section 3.2.2
 - 11: $\mathcal{F}_{Q,C} := \mathcal{F}_{Q,C} \cup \mathcal{F}_{Q,C,e}$
 - 12: Display matches of Q in $\mathcal{F}_{Q,C}$ as answer
-

3.1 Precomputations

3.1.1 Constraint Graph

A *constraint graph* is an equivalent, more compact representation of the constraints \mathcal{C} . More formally, a *constraint graph* $G_C = (V, E)$ is a pair consisting of vertices $V \subseteq Rel$ and labeled edges $E \subseteq E_u \cup E_d$. The set of edges E is in turn composed of undirected edges $E_u \subseteq V \times V \times \{mutex, disjoint\}$ and directed edges $E_d \subseteq V \times V \times \{before\}$. Thus, edges are triples consisting of two vertices (i.e., relations) that are connected by an edge with a label representing the constraint type. We remark that our notion of constraint graphs is inspired by the constraint graphs apparent in constraint satisfaction problems. See, for example, [RNC⁺96] for an introduction.

To construct the constraint graph G_C from a set of constraints \mathcal{C} , we define a bijective function $c : \mathcal{C} \rightarrow E$ as follows (relation arguments are replaced by dots):

$$c(rel_{E_1}(\cdot) \wedge rel_{E_2}(\cdot) \wedge \cdot \neq \cdot \rightarrow rel_T(\cdot)) = \begin{cases} (rel_{E_1}, rel_{E_2}, rel_T) & \text{if } rel_T \doteq disjoint \\ & \text{or } rel_T \doteq before \\ (rel_{E_1}, rel_{E_2}, mutex) & \text{if } rel_T \doteq false \end{cases}$$

It is worthwhile to accentuate that constraint graphs are solely about constraints among

relations. That is, G_C represents a higher level of abstraction than considering temporal conflicts among actual facts. It only needs to be precomputed once for a given set of constraints \mathcal{C} and can then be reused for processing an arbitrary amount of queries.

Example 5. If we apply the function c to the constraint in Formula (6), we receive the triple $(\text{bornIn}, \text{playsForClub}, \text{before})$. In Figure 2(a), the triple is indicated by the edge connecting the vertex named bornIn with playsForClub . Formulas (4) and (8) are shown in Figure 2(a) as well, both depicted as self loops, since their two relations coincide.

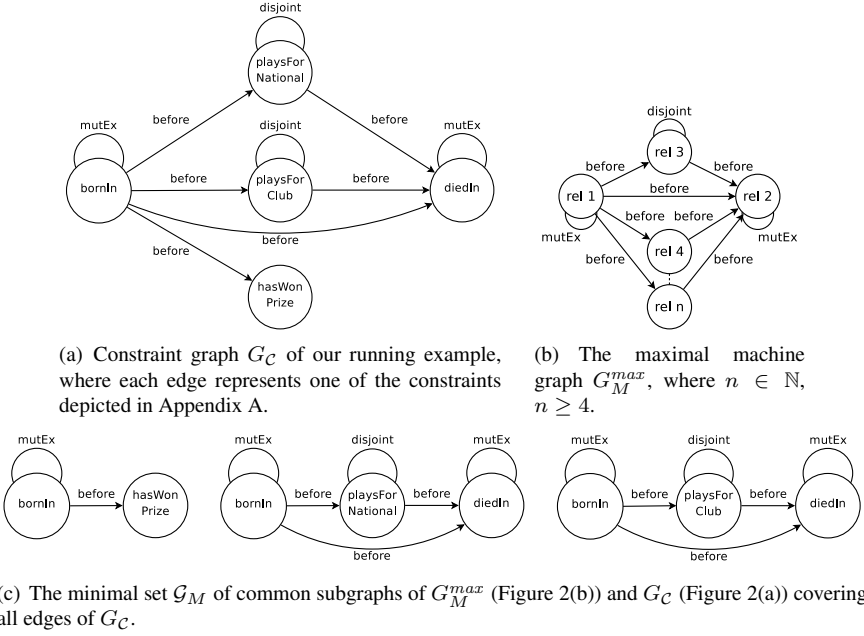


Figure 2: Graphs expressing constraints.

Constraint graphs can describe any combination of pairwise temporal constraints among relations, which might be unsatisfiable, so we focus on a subclass to be defined in the next section.

Solvable Constraint Graphs. We call a constraint graph $G_C = (V, E)$ *solvable* if its vertices can be partitioned in three sets $V = V_{begin} \dot{\cup} V_{middle} \dot{\cup} V_{end}$. Every $v \in V_{begin} \cup V_{end}$ must have exactly one loop labeled by *mutEx*, and every $v \in V_{middle}$ can have a loop labeled by *disjoint*. Furthermore, precedence edges can point from V_{begin} to $V_{middle} \cup V_{end}$ and from V_{middle} to V_{end} .

Example 6. Figure 2(a) contains a solvable constraint graph, where $V_{begin} = \{\text{bornIn}\}$, $V_{middle} = \{\text{playsForNational}, \text{playsForClub}, \text{hasWonPrize}\}$, and $V_{end} = \{\text{diedIn}\}$.

We note that solvable constraint graphs are satisfiable, as there are no cycles of precedence constraints and each pair of facts can be constrained by at most one (precedence, disjointness, or mutual-exclusion) constraint, which is the reason for limiting (3) and (7) to one extensional predicate only.

Computing Solvable Constraint Graphs. An implementation of Line 1 of Algorithm 1, which translates a set of constraints \mathcal{C} to a constraint graph G_C , can run in $O(|\mathcal{C}|)$ by iterating over the constraints, thereby creating a vertex for each relation in G_C (if not yet present), and then adding the edges as defined by the bijective function c . The condition in Line 2 of Algorithm 1 can also be implemented in $O(|\mathcal{C}|)$ by checking the following three conditions for every vertex (which are equivalent to the definition of solvable constraint graphs of the previous paragraph):

- 1) $\neg \exists rel_E \in V \text{ s.t. } (rel_E, rel_E, mutEx) \in E \wedge (rel_E, rel_E, disjoint) \in E$
- 2) $(rel_{E_1}, rel_{E_2}, before) \in E \rightarrow \left(\begin{array}{l} (rel_{E_1}, rel_{E_1}, mutEx) \in E \\ \vee (rel_{E_2}, rel_{E_2}, mutEx) \in E \end{array} \right)$
- 3) $\neg \exists rel_E, rel_{E_1}, rel_{E_2} \in V \text{ s.t. } \left(\begin{array}{l} (rel, rel, mutEx) \in E \\ \wedge (rel_{E_1}, rel_E, before) \in E \\ \wedge (rel_E, rel_{E_2}, before) \in E \end{array} \right)$

3.1.2 Machine Graphs

A machine graph corresponds to the combination of constraints to be enforced by one scheduling machine. A single scheduling machine cannot carry any combination of constraints, but at most the graph G_M^{max} displayed in Figure 2(b). Intuitively, a machine graph G_M is a subgraph of G_M^{max} or to put it differently, a scheduling machine is a part of the maximal machine.

Now, we cover a given constraint graph G_C with a set of machine graphs \mathcal{G}_M , all enclosing different combinations of constraints. As we have to respect all constraints encoded in G_C , we require that every edge in G_C is part of at least one machine graph $G_M \in \mathcal{G}_M$. Based on the scheduling machines defined by \mathcal{G}_M the scheduling algorithm in Section 3.2.2 will implement all constraints.

More formally, the set of *machine subgraphs* is a set of graphs \mathcal{G}_M which are all isomorphic to connected, vertex-induced subgraphs of both G_M^{max} and $G_C = (V_C, E_C)$. A vertex-induced subgraph is a subset of the vertices together with all the edges connecting vertices in the subset. Furthermore, we demand that $\bigcup_{(V_M, E_M) \in \mathcal{G}_M} E_M = E_C$ and that $|\mathcal{G}_M|$ is minimal in the number of subgraphs it contains. The former requirement expresses that all edges (each representing a constraint) of G_C are covered by at least one graph in \mathcal{G}_M . The latter requirement calls for a minimum number of graphs in \mathcal{G}_M , thus making scheduling more efficient.

As constraints are encoded in edges, a subgraph with no edge would be meaningless. An effect of both requirements is that subgraphs consisting of only one vertex but no edge (although being isomorphic to, for example, $rel\ 4$ in G_M^{max}) are always removed from \mathcal{G}_M , as they do not cover an edge of G_C .

Example 7. For G_C as in Figure 2(a) and G_M^{max} as in Figure 2(b), a set of common induced subgraphs covering all edges of G_C is depicted in Figure 2(c).

Computing Machine Subgraphs. The problem of finding a maximal isomorphic subgraph of two graphs is known to be NP-hard. Nevertheless, in the case of G_M^{max} , it suffices to compare the vertices $rel\ 1, \dots, rel\ 4$ with the vertices in G_C . At every comparison, we

try to expand the common subgraphs following the edges in both G_C and G_M^{max} . This is how we find one common subgraph.

To compute the full set, we aim for a minimum number of subgraphs covering all edges of G_C . If we think of the edges as elements of sets and of the subgraphs as sets, then any procedure solving the NP-hard set-cover problem can tackle our problem. For this set-cover problem, a greedy approximation algorithm, which chooses sets of maximum size first, is well established [CLRS01]. Hence we apply the same idea, by determining a maximum common subgraph with respect to the number of edges in every iteration.

3.2 Computations at Query Time

Having introduced all the precomputation steps, we move on to the procedures to be executed for each query, which builds on these precomputed data structures. Since we strive for computing a consistent set of facts, which are all relevant for answering the query, there are two major steps at query-time. The first is the retrieval of the relevant facts from a database (grounding), and the second determines the consistent subset of these facts (scheduling).

3.2.1 Grounding

One main observation is that for facts, which are not in a temporal conflict with each other, constraints do not even have to be grounded because the temporal head literal would already evaluate to *true*, such that the grounded clause would already be satisfied. Facts that do not occur in any grounded clause thus remain *true*, while only between conflicting facts, the reasoner needs to decide for a different truth assignment. Since (typically) a majority of facts is not in conflict with any other fact, this observation helps to keep the grounding phase more efficient.

Line 6 of Algorithm 1 is implemented in two steps. First, all matches to the query from the knowledge base are collected in the set \mathcal{F}_Q . Second, all facts possibly conflicting with them are added to \mathcal{F}_Q as follows. We begin by identifying all vertices in G_C corresponding to the relations of facts in the matches of the query. Then we traverse G_C in a breath-first manner starting from the identified vertices. During the traversal, we ground the occurring relations and add the retrieved facts to \mathcal{F}_Q .

A feature of G_C is that every connected component shares the first argument resulting from (1). Hence we have to execute a breath-first traversal for every member in $FirstArg$, which results in an implementation with $O(|G_C| \cdot |FirstArg|)$ run-time.

Example 8. Let Q be from (2), G_C from Figure 2(a), and \mathcal{F} from Figure 1. The initial matches of Q are $\mathcal{F}_Q = \{f_{playsBMU}, f_{playsBB}\}$. So $FirstArg = \{David_Beckham\}$, which means there is only one traversal. We start from *playsForClub*, visit *bornIn* and *diedIn* in the first stage, and finally *playsForNational* and *hasWonPrize*. So, f_{bornBL} and $f_{bornBOT}$ are added to \mathcal{F}_Q first, followed by $f_{playsBE}$, which results in $\mathcal{F}_Q = \mathcal{F}$.

3.2.2 Scheduling Problem

Once we have retrieved all relevant facts \mathcal{F}_Q , we continue by identifying a maximum-weight consistent subset of the facts $\mathcal{F}_{Q,C}$. We map this problem to a scheduling problem, consisting of *scheduling machines* and *scheduling jobs*.

- A *scheduling machine* is a time interval of \mathcal{T} with a *capacity* $\in \mathbb{R}^+$.
- A *scheduling job* is a weighted time interval of \mathcal{T} coming with different sizes for each machine, i.e., $size : Jobs \times Machines \rightarrow [0, capacity]$.

We note that all scheduling machines share the same *capacity*.

A *scheduling problem* is a set of scheduling machines *Machines* and a set of scheduling jobs *Jobs*, where the task is to find a subset $J' \subseteq Jobs$ of jobs which maximize the sum of weights

$$\max_{J' \subseteq Jobs} \sum_{j \in J'} weight(j) \cdot x_j$$

such that

$$\forall m \in Machines, \forall t \in \mathbb{N}_0 \quad \sum_{j \in J' | begin(j) \leq t < end(j)} size(j, m) \cdot x_j \leq capacity$$

and $x_j \in \{0, 1\}$.

In words, we are looking for a maximum-weight subset of the jobs, such that the capacity of each machine is not exceeded by the sum of the sizes of the jobs running on them. The variable x_j indicates whether the job belongs to the solution ($x_j = 1$) or not ($x_j = 0$).

We remark, that the above optimization problem is NP-hard, as we obtain the Knapsack problem as a special case, i.e., by considering only one scheduling machine for all constraints and one time interval $[0, +\infty)$ for all facts.

Mapping Constraint Graphs to Scheduling Machines. Next, we map the search for a consistent subset of facts to the above scheduling problem by relating every fact in \mathcal{F}_Q with a scheduling job and every graph in \mathcal{G}_M with a scheduling machine. To encode a conflict between two facts in the scheduling problem, we ensure that the intervals of the corresponding jobs are overlapping, and there is at least one machine which cannot process both jobs at the same time.

We begin with the assignment of different sizes to facts on different machines as defined by the function $size : \mathcal{F}_Q \times \mathcal{G}_M \rightarrow [0, capacity]$ where

$$size(\underbrace{f_{rel}}_{\in \mathcal{F}_Q}, \underbrace{(V, E)}_{\in \mathcal{G}_M}) = \begin{cases} 0 & \text{if } rel \notin V \\ capacity & \text{if } rel \in V \text{ and } rel \text{ represented by} \\ & \text{'rel 1' or 'rel 2' in } G_M^{max} \\ \frac{capacity}{2} + \epsilon & \text{if } rel \in V \text{ and } rel \text{ represented by 'rel 3' in } G_M^{max} \\ \frac{\frac{capacity}{2} - \epsilon}{|\mathcal{F}_Q|} & \text{if } rel \in V \text{ and } rel \text{ represented by 'rel 4' in } G_M^{max} \end{cases}$$

and we use f_{rel} to denote a fact with relation rel .

If a fact is not constrained by $G_M \in \mathcal{G}_M$, we set its size to zero, so no conflicts result. Second, if a fact is an instance of vertices *rel* 1 or *rel* 2, then it is subject to a mutual exclusion constraint. Hence, the size is fixed to *capacity*, which makes its job mutually exclusive to all overlapping jobs of non-zero size. In the third case, by assigning $\frac{\text{capacity}}{2} + \epsilon$ (for an $\epsilon > 0$) to the size of the fact (job), we achieve that all facts of *rel* 3 become mutually exclusive if they overlap. Finally, the fourth case sets the size of jobs corresponding to facts matching *rel* 4 in G_M^{\max} to $\frac{\frac{\text{capacity}}{2} - \epsilon}{|\mathcal{F}_Q|}$, which admits all of them to be scheduled even though a job related to case three is scheduled at the same time.

The above construction models disjointness correctly, but it fails for precedence and mutual-exclusion. For example, two facts, which are supposed to be mutually exclusive but have no overlap in their intervals, could be scheduled.

So we continue with the translation from intervals of facts to intervals of jobs as defined by the functions $\text{begin} : \mathcal{F} \times 2^{\mathcal{G}_M} \rightarrow \mathbb{N}_0$ and $\text{end} : \mathcal{F} \times 2^{\mathcal{G}_M} \rightarrow \mathbb{N}_0 \cup \{+\infty\}$ where,

$$\text{begin}(f_{\text{rel}, [t_b, t_e]}, \mathcal{G}_M) = \min\{t_b\} \cup \left\{ 0 \mid \exists G_M \in \mathcal{G}_M. \begin{array}{l} G_M = (V, E), \text{rel} \in V, \\ \text{rel isomorphic to rel 1 in } G_M^{\max} \end{array} \right\}$$

and

$$\text{end}(f_{\text{rel}, [t_b, t_e]}, \mathcal{G}_M) = \max\{t_e\} \cup \left\{ +\infty \mid \exists G_M \in \mathcal{G}_M. \begin{array}{l} G_M = (V, E), \text{rel} \in V, \\ \text{rel isomorphic to rel 2 in } G_M^{\max} \end{array} \right\}$$

and we use $f_{\text{rel}, [t_b, t_e]}$ to represent a fact with relation *rel* and interval $[t_b, t_e]$. Again, the weight $w(j)$ of a scheduling job j is simply the weight $w(f)$ of the associated fact f .

Both functions leave all interval limits of facts not being subject of a mutual-exclusion constraint untouched. On the contrary, the interval limit is either set to the very begin or the very end, depending on the possible precedence constraints. As a result, all intervals of mutual-exclusive facts overlap either in 0 or $+\infty$. At the same time, facts of *rel* 1 cannot be preceded by other facts, as they start at 0, thus correctly modeling precedence. A symmetric argument holds for instances of *rel* 2.

Computing the Mapping. Regarding complexity, the mapping from a set of facts $|\mathcal{F}_Q|$ to the corresponding scheduling jobs can be done in $O(|\mathcal{F}_Q|)$, since we can compute the mapping for each fact independently by applying the functions *size*, *begin*, and *end*.

$f \in \mathcal{F}$	$\text{size}(f, \text{left})$	$\text{size}(f, \text{middle})$	$\text{size}(f, \text{right})$	$\text{begin}(f, \text{all})$	$\text{end}(f, \text{all})$
f_{bornBL}	<i>capacity</i>	<i>capacity</i>	<i>capacity</i>	0	1976
f_{bornBOT}	<i>capacity</i>	<i>capacity</i>	<i>capacity</i>	0	2000
f_{playsBMU}	0	0	$\frac{\text{capacity}}{2} + \epsilon$	1993	2004
f_{playsBB}	0	0	$\frac{\text{capacity}}{2} + \epsilon$	1999	2001
f_{playsBE}	0	$\frac{\text{capacity}}{2} + \epsilon$	0	1992	2011

Table 1: The translation of the facts \mathcal{F} of Figure 1 to scheduling jobs using *capacity* = 1.0, where the second argument of *size* and *end* refer to the graphs of Figure 2(c).

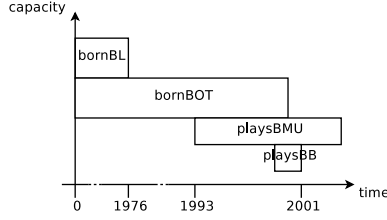


Figure 3: Jobs (translated facts) of Table 1 for the scheduling machine (graph) at the right of Figure 2(c).

Example 9. The translation of the facts of Figure 1 to three scheduling machines with respect to the graph \mathcal{G}_M of Figure 2(c) is shown in Table 1. Additionally, Figure 3 depicts the facts f_{bornBL} , f_{bornBOT} , f_{playsBMU} , and f_{playsBB} to be scheduled on the machine corresponding to the graph at the right of Figure 2(c).

Computing a Consistent Subset. Algorithm 2 presents an efficient approximation algorithm for the NP-hard scheduling problem, whose performance is analyzed empirically by the experiments in Section 4. It is inspired by the general scheduling framework presented in [BNBYF⁺01].

Every connected component of a solvable constraint graph G_C shares one variable as both relations in (1) have the same variable as their first argument. As a result, only facts with identical entities as their first argument can be in conflict. Thus, we invoke Algorithm 2 for every entity $e \in \text{FirstArg}$ (see Lines 8 to 11 in Algorithm 1).

Algorithm 2 is based on the interplay with a stack and consists of a pushing phase (Lines 3 to 10) during which some facts are pushed onto the stack, and a popping phase (Lines 12 to 17) during which facts are popped from the stack and possibly included in the solution. In the first step of the pushing phase, the fact f with minimum $\text{end}(f, G_C)$ is pushed onto the stack, while the weight of every interval in conflict with f is decreased by $w(f)$. Intervals with negative weights are then removed and ignored from further consideration. In the next step, the fact whose end time is minimal among the remaining ones is pushed onto the stack, while the weights of its conflicting facts are decreased and all facts with negative weights are removed. These steps are iterated until every fact is either on the stack or is deleted. In the popping phase, facts are iteratively popped from the stack and included in the solution if this maintains feasible, or—in the scheduling sense—if the fact does fit on the machines. The algorithm ends when the stack becomes empty.

The worst-case complexity of Algorithm 2 is $O(|\mathcal{F}_{Q,\epsilon}|^2 |\mathcal{G}_M|)$, which is dominated by the three nested loops in Lines 3 to 5. After the example, we will explain how to improve this worst-case run-time, while we keep Algorithm 2 for its easier presentation.

Example 10. We execute Algorithm 2 for the problem setting of Figure 3, where we assume $\epsilon = 0.1$ and $\text{capacity} = 1.0$. The loop in Line 3 inspects the facts ordered by end as f_{bornBL} , f_{bornBOT} , f_{playsBB} , and f_{playsBMU} , where only f_{bornBOT} does not get pushed to the stack as its weight becomes negative in a conflict with f_{playsBB} . Continuing with the loop in Line 12 we schedule first f_{playsBMU} , then we omit f_{playsBB} , be-

cause it exceeds the capacity at from 1999 to 2001. Finally, f_{bornBL} is added, such that $\mathcal{F}_{Q,C,e} = \{f_{\text{playsBMU}}, f_{\text{bornBL}}\}$.

Algorithm 2 Resolving conflicts

Require: A set of facts $\mathcal{F}_{Q,e}$ with identical first argument e

Require: A machine set \mathcal{G}_M

```

1: Initialize a stack  $\mathcal{S} = \langle \rangle$ 
2: Sort all  $f \in \mathcal{F}_{Q,e}$  by  $\text{end}(f, \mathcal{G}_M)$ 
3: for all  $f \in \mathcal{F}_{Q,e}$  by increasing  $\text{end}(f, \mathcal{G}_M)$  do
4:   for all machine graphs  $G_M \in \mathcal{G}_M$  do
5:     for all  $f' \in \mathcal{S}$  do
6:       if  $f$  and  $f'$  intersect and  $\text{size}(f, G_M) > 0$ ,  $\text{size}(f', G_M) > 0$  then
7:          $w(f') := w(f') - \text{size}(f', G_M) \cdot w(f)$ 
8:         if  $w(f') \leq 0$  then
9:           Remove  $f'$  from  $\mathcal{S}$ 
10:   Push  $f$  to  $\mathcal{S}$ 
11:  $\mathcal{F}_{Q,C,e} := \emptyset$   $\triangleright \mathcal{F}_{Q,C,e} \subseteq \mathcal{F}_{Q,e}$ 
12: while  $\mathcal{S}$  is not empty do
13:    $f_{[t_b, t_e)} := \mathcal{S}.\text{pop}()$ 
14:   for all  $G_M \in \mathcal{G}_M$  do
15:     if  $\forall t \in [t_b, t_e). \text{capacity}_{\text{used}}(G_M, t) + \text{size}(f, G_M) > \text{capacity}$  then
16:       Continue with loop in Line 12
17:   Add  $f_{[t_b, t_e)}$  to  $\mathcal{F}_{Q,C,e}$ 
18:   for all  $G_M \in \mathcal{G}_M$  do
19:      $\forall t \in [t_b, t_e). \text{capacity}_{\text{used}}(G_M, t) := \text{capacity}_{\text{used}}(G_M, t) - \text{size}(f, G_M)$ 
20: return  $\mathcal{F}_{Q,C,e}$   $\triangleright \mathcal{F}_{Q,C,e} \subseteq \mathcal{F}_{Q,e}$ 

```

Improving the Worst-Case Complexity. Following Section 3.3 of [BNBYF⁺01], the worst-case complexity can be reduced to $O(|\mathcal{F}_{Q,e}| \log |\mathcal{F}_{Q,e}| + |\mathcal{F}_{Q,e}| |\mathcal{G}_M|)$, thus breaking the quadratic barrier and allowing us to efficiently process huge sets of conflicting facts.

The main idea is to replace the stack of intervals by a sorted list of interval end-times (for both begin and end). Then the pushing-phase is substituted by a forward-iteration over the list. The weight of the intersecting intervals can be obtained implicitly by keeping track of the total amount of weights of the iterated intervals and by comparing this value at both end-times of the intervals. In a similar manner, the popping phase is changed to a backwards-iteration over the list. In total, both iterations for each graph in \mathcal{G}_M require $O(|\mathcal{F}_{Q,e}| |\mathcal{G}_M|)$ steps, where we have to add $O(|\mathcal{F}_{Q,e}| \log |\mathcal{F}_{Q,e}|)$ steps in order to create the sorted list of interval end-times.

4 Experiments

System. Our system featuring the algorithms of the previous section was implemented in Java 1.6 in about 3k lines of code. As a back-end, a Postgres 8.3 database is deployed to

store the RDF triples along with their corresponding weights and time intervals. Both the program and the database are run on the same Intel E8200 machine with 4 GB RAM.

Competitors. We can reduce the optimization problem of Section 2.5 to the Maximum Weight Independent Set problem (MWIS)² by considering facts as vertices and drawing an edge between them if they are in conflict. Then a maximum-weight subset of vertices (facts), that do not share an edge (according to the definition of MWIS), coincides with a conflict-free solution. Thus, we utilize a simple exponential time algorithm to compute the optimal solution of MWIS as long as this remains feasible.

Additionally, we employ a greedy heuristic [BSK10] for the MWIS, which proved to perform best on our data among all the greedy methods we tried. There are other means of approximating the MWIS problem, like stochastic optimization. However they are even less scalable than greedy methods [BBPP99]. As the greedy methods are based on the graph, the ingredients for choosing a fact (vertex), in order to remove or add facts to the approximated MWIS, are the weights of the facts (vertices) and the number of conflicting facts (degree of the vertex). Thus, the worst-case run-time is in $\Omega(|\mathcal{F}_Q|^2)$, as there can be quadratically many edges. Hence, in terms of run-time complexity, our scheduling algorithm also asymptotically performs better than this greedy approach, as it is based on sorting facts (vertices) represented by scheduling jobs, rather than enumerating all pairs of facts (edges), which are in conflict with each other.

Parameters, Constraints & Queries. The only free parameter is $0.5 > \epsilon > 0$ (Section 3.2) which we fixed to $\epsilon = 0.49$, as we have good experiences with values close to 0.5. As constraints, we employ the formulas of Appendix A, and as query we use Equation (2).

Dataset. T-YAGO [WZQ⁺10] contains data about the *playsForClub*, *playsForNational*, and *hasWonPrize* relations, which we extended manually by dates of birth and death. Nevertheless, the data in T-YAGO is nearly conflict-free, thus we add synthetic facts to create conflicts in the following manner.

First, we choose one of the consistent facts uniformly. Then we create a perturbed copy by drawing the start-time of the interval, the length of the interval, and the confidence from three different Gaussians $\mathcal{N}(\mu_s, \sigma_s^2)$, $\mathcal{N}(\mu_l, \sigma_l^2)$, and $\mathcal{N}(\mu_c, \sigma_c^2)$, respectively. The means μ_s , μ_l , and μ_c are set to the original value of the fact contained in T-YAGO, whereas the variances are varied during the experiments to produce problem instances of diverse nature (see Figure 4(a)). By writing n , we refer to the number of added synthetic facts about the queried entity.

Approximation Ratio. In order to evaluate the performance of the algorithms, we define the approximation ratio as $\frac{W}{W^*}$, where W and W^* represent the sum of the weights computed by a heuristic and the optimal exponential-time algorithm, respectively.

Results. Our algorithm showed impressive robustness with respect to the perturbed data as shown in Figure 4(a). In particular, its average approximation ratio never dropped below 0.98. In Figure 4(b) we show the distribution of approximation ratios for 1,000 runs, whereas the previous three figures focused on the mean. The histogram of our scheduling algorithm exhibits excellent behavior as in nearly every problem instance the optimal so-

²The opposite direction compared to the reduction in the hardness paragraph of Section 2.5.

lution was found. The greedy heuristic for MWIS does little worse, but still is very good. The run-time of the scheduling algorithm and the grounding algorithm (both described in Section 3.2) is depicted in the left of Figure 4(c). Their complexities are sub-quadratic. Finally, the run-times of the MWIS greedy heuristic and its grounding procedure are displayed in the right of Figure 4(c). Admittedly, the implementations were less optimized, however optimization can only lower the constants, but not the quadratic complexity.

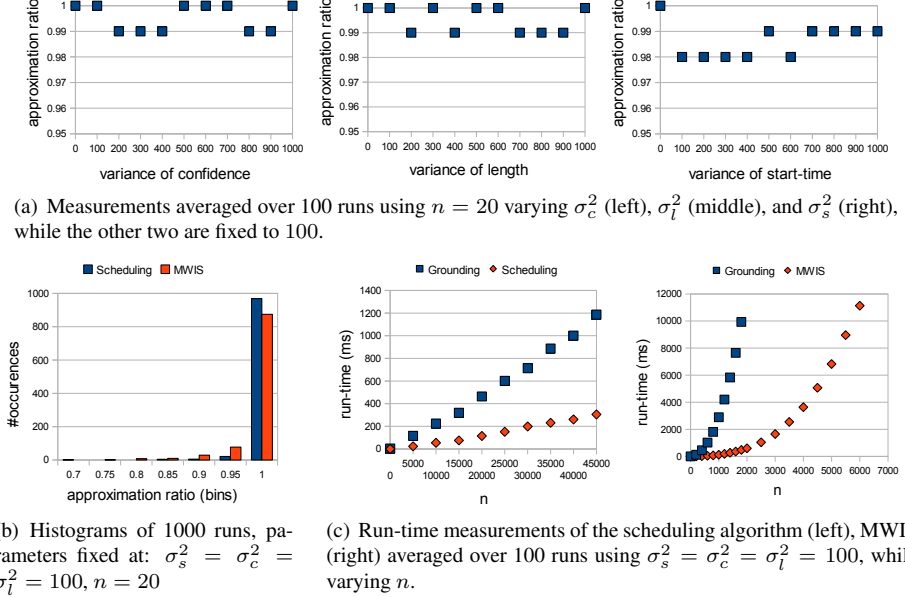


Figure 4: Experiments

5 Related Work

Temporal RDF. Temporal databases were introduced more than 25 years ago [JS99]. Early work on RDF and time, which discusses many design issues, can be found in [GHV05], and which was later pursued in [GHV07]. A query language for RDF with temporal capabilities was presented in [TB09], which is a complementary issue compared to our work. Moreover, [PUS08] introduces an indexing scheme for time-annotated RDF triples without confidence values. Its notion of consistency rejects contradicting statements about the number of validity points in a time interval, whereas its temporal distance metric is purely used for indexing purposes.

Temporal Constraints. The relations between temporal intervals probably were first introduced in [All83] and were later extended in various ways, where [FGV05] provides a comprehensive overview. Additionally, [FGV05] contains an outline of how to encode time in first-order logic. In terms of Description Logics, there are several temporal extensions, where [AF00, LWZ08] provide surveys. Temporal Constraint Satisfaction problems

[SV98] are usually not based on data but focus on the search for a valid solution in terms of variables representing time which fulfill given constraints. Regarding temporal constraints on RDF graphs, purely theoretical work was carried out in [HV06].

Machine Learning. In the machine learning community, there exist frameworks [RY05] and [RD06] for supporting general constraints on uncertain data whose performances are rather slow compared to our algorithm, due to solving general ILP problems and the grounding algorithm solely being based on typing, respectively.

Scheduling. Intensive research was conducted in the scheduling field with numerous applications [Pin08, LKA04]. Still, the combination of precedence and disjointness constraints is not well covered, and to our best knowledge, only [XP90] presents an algorithm tackling the problem. Yet, its limited scalability makes it unsuitable for bigger data sets.

Maximum Weight Independent Set. In the past, many heuristics for the MWIS problem [BBPP99, JT96] have been developed, covering—among others—greedy approaches, stochastic optimization like simulated annealing or genetic algorithms, and hybrid methods of these. However, our implicit representation of conflicts (see Section 3.2.2, last paragraph) is more scalable than the explicit form using edges of a graph.

Uncertain and Probabilistic Databases. Recent work on uncertain data management and probabilistic databases [OSH⁺08, AJKO08, DS07], including our own work [DSTW08, DSTW10], have shown how to represent and handle dependencies of data objects inside an SQL-like environment. Yet, only very few database-oriented works on handling temporal inconsistencies in a first-order reasoning setting have been proposed so far. In [WYT10], we devised a probabilistic model, based on time histograms and data lineage, for a first-order, rule-based reasoner with temporal predicates. The rules considered in that work do not consider the inclusion of actual consistency constraints, where only some facts out of a given set may be set to true while other facts are considered false. Technically, this resolves to including also negation into the constraints, while [WYT10] considers positive lineage (i.e., conjunctions and disjunctions) only. Moreover, our approach resembles some similarity to probabilistic extensions to Datalog [Fuh95], however, no resolution of inconsistencies or forms of temporal reasoning had been considered in this context.

6 Conclusions

We have presented a declarative framework for temporal consistency reasoning in uncertain and inconsistent knowledge bases. Our approach works by identifying a subclass of first-order consistency constraints, which can be efficiently mapped to constraint graphs and be solved using results from scheduling theory. Our experiments show that our approach performs superior to common approximation heuristics that directly operate over the underlying Maximum Weight Independent Set problem in terms of both run-time and approximation quality. As for future work, we aim to investigate in further generalizing the class of constraints we can solve with our approach, and we also aim at making our interval operations more fine-grained, for example, by cutting off conflicting intervals, or by incorporating time histograms that may capture different confidences in a fact's validity at different points in time.

Acknowledgments: We would like to thank Yafang Wang, Mohamed Yahya, and Gerhard Weikum for providing the temporal data of T-YAGO for our experiments and for their helpful discussions. We also thank the reviewers for their helpful comments.

References

- [AF00] A. Artale and E. Franconi. A survey of temporal extensions of description logics. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):171–210, 2000.
- [AJKO08] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and Simple Relational Processing of Uncertain Data. In *ICDE*, pages 983–992, 2008.
- [All83] J. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.
- [BBPP99] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo. The Maximum Clique Problem. In *Handbook of combinatorial optimization*, pages 1–174. Kluwer, 1999.
- [BNBYF⁺01] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber. A unified approach to approximating resource allocation and scheduling. *J. ACM*, 48(5):1069–1090, 2001.
- [BSK10] S. Balaji, V. Swaminathan, and K. Kannan. A Simple Algorithm to Optimize Maximum Independent Set. *Advanced Modeling and Optimization*, 12(1):107–118, 2010.
- [CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, second edition, July 2001.
- [DS07] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.
- [DSTW08] A. Das Sarma, M. Theobald, and J. Widom. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *ICDE*, pages 1023–1032, 2008.
- [DSTW10] A. Das Sarma, M. Theobald, and J. Widom. LIVE: A Lineage-Supported Versioned DBMS. In *SSDBM*, volume 6187 of *LNCS*, pages 416–433, 2010.
- [FGV05] M. Fisher, D. Gabbay, and L. Vila. *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier, 2005.
- [Fuh95] N. Fuhr. Probabilistic Datalog - A Logic For Powerful Retrieval Methods. In *SIGIR*, pages 282–290, 1995.
- [GHV05] C. Gutiérrez, C. Hurtado, and A. Vaisman. Temporal RDF. In *ESWC*, volume 3532 of *LNCS*, pages 93–107, 2005.
- [GHV07] C. Gutiérrez, C. Hurtado, and A. Vaisman. Introducing Time into RDF. *IEEE Trans. on Knowl. and Data Eng.*, 19(2):207–218, 2007.
- [GS93] M. C. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: a graph-theoretic approach. *J. ACM*, 40(5):1108–1133, 1993.
- [HV06] C. Hurtado and A. Vaisman. Reasoning with Temporal Constraints in RDF. In *PPSWR Workshop*, volume 4187 of *LNCS*, pages 164–178, 2006.

- [JS99] C. Jensen and R. Snodgrass. Temporal Data Management. *IEEE Trans. on Knowl. and Data Eng.*, 11(1):36–44, 1999.
- [JT96] D. Johnson and M. Trick, editors. *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS*, 1996.
- [LKA04] J. Leung, L. Kelly, and J. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, 2004.
- [LWZ08] C. Lutz, F. Wolter, and M. Zakharyashev. Temporal Description Logics: A Survey. In *TIME*, pages 3–14, 2008.
- [OSH⁺08] B. Omar, A. Das Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.
- [Pin08] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, third edition, 2008.
- [PUS08] A. Pugliese, O. Udre, and V. S. Subrahmanian. Scaling RDF with Time. In *WWW*, pages 605–614, 2008.
- [RD06] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.
- [RNC⁺96] S. Russell, P. Norvig, J. Candy, J. Malik, and D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, 1996.
- [RY05] D. Roth and W. Yih. Integer Linear Programming Inference for Conditional Random Fields. In *ICML*, pages 737–744, 2005.
- [SV98] E. Schwalb and L. Vila. Temporal Constraints: A Survey. *Constraints*, 3(2/3):129–149, 1998.
- [TB09] J. Tappolet and A. Bernstein. Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In *ESWC*, pages 308–322. Springer, 2009.
- [WYT10] Y. Wang, M. Yahya, and M. Theobald. Time-aware Reasoning in Uncertain Knowledge Bases. In *MUD Workshop*, 2010.
- [WZQ⁺10] Y. Wang, M. Zhu, L. Qu, M. Spaniol, and G. Weikum. Timely YAGO: harvesting, querying, and visualizing temporal knowledge from Wikipedia. In *EDBT*, 2010.
- [XP90] J. Xu and D. Parnas. Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations. *IEEE Trans. Softw. Eng.*, 16(3):360–369, 1990.

A Constraints Used for Experiments

$$\begin{aligned}
&(\text{bornIn}(p, l_1, t_1) \wedge \text{bornIn}(p, l_2, t_2) \wedge l_1 \neq l_2) \rightarrow \text{false} \\
&(\text{bornIn}(p, l_1, t_1) \wedge \text{diedIn}(p, l_2, t_2)) \rightarrow \text{before}(t_1, t_2) \\
&(\text{bornIn}(p, l, t_1) \wedge \text{playsForClub}(p, c, t_2)) \rightarrow \text{before}(t_1, t_2) \\
&(\text{bornIn}(p, l, t_1) \wedge \text{playsForNational}(p, n, t_2)) \rightarrow \text{before}(t_1, t_2) \\
&(\text{bornIn}(p, l, t_1) \wedge \text{hasWonPrize}(p, pr, t_2)) \rightarrow \text{before}(t_1, t_2) \\
&(\text{playsForNational}(p, n_1, t_1) \wedge \text{playsForNational}(p, n_2, t_2) \wedge c_1 \neq c_2) \rightarrow \text{disjoint}(t_1, t_2) \\
&(\text{playsForClub}(p, c_1, t_1) \wedge \text{playsForClub}(p, c_2, t_2) \wedge c_1 \neq c_2) \rightarrow \text{disjoint}(t_1, t_2) \\
&(\text{playsForClub}(p, c, t_1) \wedge \text{diedIn}(p, l, t_2)) \rightarrow \text{before}(t_1, t_2) \\
&(\text{playsForNational}(p, n, t_1) \wedge \text{diedIn}(p, l, t_2)) \rightarrow \text{before}(t_1, t_2) \\
&(\text{diedIn}(p, l_1, t_1) \wedge \text{diedIn}(p, l_2, t_2) \wedge l_1 \neq l_2) \rightarrow \text{false}
\end{aligned}$$

QSQL^p: Eine Erweiterung der probabilistischen Many-World-Semantik um Relevanzwahrscheinlichkeiten

Sebastian Lehrack, Sascha Saretz und Ingo Schmitt

Brandenburgische Technische Universität Cottbus
Institut für Informatik, Postfach 10 13 44
D-03013 Cottbus, Germany
{slehrack,sascha.saretz}@informatik.tu-cottbus.de,
schmitt@tu-cottbus.de

Zusammenfassung. Die traditionelle Auswertung einer Datenbankanfrage ermittelt für jedes Tupel entweder den Wahrheitswert *Wahr* oder *Falsch*. Für viele Anwendungsszenarien ist diese Auswertungssemantik zu restriktiv, insbesondere wenn ein differenzierteres Anfrageergebnis benötigt wird. Ein etablierter probabilistischer Ansatz zum Erreichen dieser Ausdifferenzierung ist die Verwendung sogenannter Relevanzwahrscheinlichkeiten: Mit welcher Wahrscheinlichkeit ist ein Dokument oder ein Datenobjekt bezüglich einer gestellten Anfrage relevant?

Neben den IR-motivierten Relevanzwahrscheinlichkeiten hat sich in der Datenbankforschung das Gebiet der probabilistischen Datenbanken etabliert. Auch hier wird ein striktes, deterministisches Auswertungsmodell als nicht mehr ausreichend angesehen. In probabilistischen Datenbanksystemen werden daher mehrere mögliche Zustände für ein und dasselbe System in einer gemeinsamen Datenbank verwaltet.

Die vorliegende Arbeit verbindet diese beiden probabilistischen Ansätze zu einem semantisch reicheren Anfragemodell.

1 Motivation

Die traditionelle Auswertung einer Datenbankanfrage ermittelt für jedes Tupel entweder den Wahrheitswert *Wahr* oder *Falsch*. Alle wahren Tupel bilden daraufhin die Ergebnismenge der Anfrage. Für viele Anwendungsszenarien ist diese Auswertungssemantik zu restriktiv, insbesondere wenn ein differenziertes Anfrageergebnis benötigt wird. Die Ausdifferenzierung des Ergebnisses setzt oft eine Aussage über den Grad der Erfüllung einer gestellten Anfrage voraus. Ein etablierter Ansatz, welcher vor allem im Bereich des Information Retrievals weit verbreitet ist, drückt den Erfüllungsgrad mittels sogenannter Relevanzwahrscheinlichkeiten aus [20]: *Mit welcher Wahrscheinlichkeit wird ein Dokument oder ein Datenobjekt bezüglich einer gestellten Anfrage vom Anwender als relevant eingestuft?* Die Entscheidung, ob ein betrachtetes Dokument oder Datenobjekt für den Anwender relevant oder nicht relevant ist, wird in dem hier betrachteten Kontext in den Erfüllungsgrad einer logikbasierten Anfrage übertragen. Ein

zentraler Bestandteil dieser Art von Anfragen sind Ähnlichkeitsprädikate, z.B. *Preis möglichst um 100 Euro* oder *Ort nahe Cottbus*, deren reelle Auswertungsergebnisse aus dem Intervall $[0; 1]$ als Relevanzwahrscheinlichkeiten interpretiert werden können.

Neben den Relevanzwahrscheinlichkeiten aus dem Bereich des Information Retrievals hat sich in der Datenbankforschung ebenfalls das Gebiet der probabilistischen Datenbanken etabliert. Auch hier wird ein striktes, deterministisches Auswertungsmodell als nicht mehr ausreichend angesehen. Insbesondere wenn Daten automatisch extrahiert werden oder aus verschiedenen Quellen stammen, existiert oft eine Unsicherheit über die Genauigkeit der so gewonnenen Daten. Neben der Unsicherheit von Daten sind menschliche Bewertungen oder Beobachtungen, welche auf Grund ihrer inhärenten Subjektivität oft mit einem Konfidenzwert annotiert werden, ein typisches Anwendungsfeld für probabilistische Datenbanken. Das vorherrschende Anfrage- und Datenmodell ist dabei die sogenannte *Many-World-Semantik*. Hier werden mehrere mögliche Zustände für ein und dasselbe System in einer gemeinsamen Datenbank verwaltet.

Die vorliegende Arbeit verbindet diese beiden probabilistischen Ansätze zu einem semantisch reicheren Anfragemodell. Insbesondere liefert sie Beiträge zu folgenden Schwerpunkten:

- die Erweiterung der Many-World-Semantik um Relevanzwahrscheinlichkeiten in einem erweiterten probabilistischen Anfragemodell,
- das Konzept einer differenzierten Normalisierung von probabilistischen Anfragen, sowie
- die praktische Umsetzung des entwickelten probabilistischen Anfragemodells durch die SQL-Erweiterung QSQL^P.

In den sich anschließenden Kapiteln sollen Beispielanfragen aus einem durchgängigen Szenario betrachtet werden. Das hier verwendete Beispielszenario beschäftigt sich mit der Beobachtung von Vögeln (Ornithologie). Hierfür werden die beiden Relationen *VBeob* (Vogelbeobachtung, siehe Abb. 1) und *VArt* (Vogelart, siehe Abb. 2) eingeführt. Für jedes Tupel der Relation Vogelbeobachtung ist ein individueller Konfidenzwert hinterlegt (Attribut *Pr*). Dagegen sind in der Relation Vogelart einzelne Eigenschaften, wie die Verbreitungsregion (Attribut *Region*) und ein charakteristisches Foto (Attribut *Bild*) der jeweiligen Vogelart abgespeichert.

2 Anfragemodelle

Um die graduelle Erfüllung von Anfragen zu ermöglichen wurden in der Vergangenheit verschiedene Ansätze entwickelt, so z.B. die Fuzzy Logik [22] von Zadeh, eine Vielzahl probabilistischer Verfahren (siehe Kapitel 6) und ein quantenlogisches Auswertungsmodell von Schmitt [18]. In diesem Kapitel soll gezeigt werden, wie die Anfrageergebnisse des quantenlogischen Auswertungsmodells als Relevanzwahrscheinlichkeiten interpretiert werden können, um diese anschließend mit der Many-World-Semantik zu kombinieren.

<i>VBeob</i> (Vogelbeobachtung)			
<i>Art</i>	<i>Ort</i>	<i>Zeit</i>	<i>Pr</i>
Star	Cottbus	September	0.9
Fink	Berlin	Juni	0.5
Amsel	Cottbus	Mai	0.4
Star	Cottbus	August	0.3
Drossel	Berlin	Juni	0.4

Abb. 1. Relation *VBeob*

<i>VArt</i> (Vogelart)		
<i>Art</i>	<i>Region</i>	<i>Bild</i>
Star	Mitteldeutschland	\square_1
Fink	Norddeutschland	\square_2
Amsel	Mitteldeutschland	\square_3
Star	Süddeutschland	\square_4

Abb. 2. Relation *VArt*

2.1 Relevanzwahrscheinlichkeiten im quantenlogischen Auswertungsmodell

Im Folgendem wird eine kurze Einführung in die Arbeitsweise des quantenlogischen Auswertungsmodells gegeben. Für eine tiefergehende Darstellung wird auf [10] und [18] verwiesen, wobei in [10] lediglich ein mathematisches Grundverständnis vorausgesetzt wird.

Die Grundidee dieses Ansatzes ist die Anwendung eines mathematischen Vektorraummodells aus der Quantenmechanik und -logik. Die abgefragten Tupel, sowie die gestellte Anfrage werden dabei als Bestandteile dieses Vektorraums modelliert. So werden z.B. die Attributwerte des abgefragten Tupels in die *Richtung eines normierten Vektors* abgebildet. Die gestellte Anfrage erzeugt dagegen ein eingebetteten Vektorunterraum, welcher auch als *Anfrageraum* bezeichnet wird. Der Anfrageraum verkörpert die gesamte Anfragesemantik. Das Auswertungsergebnis wird dann durch den minimalen einschließenden Winkel zwischen Tupelvektor und Anfrageraum bestimmt. Dabei bedeutet ein Winkel von 0° eine maximale Ähnlichkeit und ein Winkel von 90° repräsentiert eine maximale Unähnlichkeit zwischen dem betrachteten Tupel und der formulierten Anfrage. Setzt man den Winkel in die quadrierte Kosinus-Funktion ein, ergibt sich ein reeller Wert zwischen 0 (für 90°) und 1 (für 0°). Dieser Wert, welcher auch als *Score-Wert* bezeichnet wird, kann demnach als Ähnlichkeitsmaß interpretiert werden.

Neben dieser geometrischen Deutung existiert eine weitere Interpretation für den berechneten Score-Wert. Die Berechnung des Score-Wertes bezüglich eines Anfrageraumes genügt den Eigenschaften eines additiven Wahrscheinlichkeitsmaßes [11].

Damit drückt der Score-Wert aus, wie wahrscheinlich es ist, dass der betrachtete Tupelvektor komplett im angefragten Anfrageraum liegt. In diesem Fall würde ein einschließender Winkel von 0° und ein Score-Wert von $\cos^2(0^\circ) = 1$ vorliegen. Somit kann der Score-Wert auch als Relevanzwahrscheinlichkeit eines Tupels gegenüber einer Anfrage aufgefasst werden, was voraussetzt, dass die komplette Erfüllung der Anfrage den betrachteten Tupels als relevant einstuft. Das Wahrscheinlichkeitsmaß wird dabei über die Konstruktion des Tupelvektors und des Anfragevektorraumes definiert.

Interessanterweise kann die Berechnung der Relevanzwahrscheinlichkeiten für ein Tupel t , die in unserem Modell eine geometrische Interpretation besitzen, auf

die logische Struktur einer Anfrage c und die Anwendung der bekannten Aggregationsfunktionen für Wahrscheinlichkeiten unabhängiger Ereignisse zurückgeführt werden:

$$\begin{aligned} eval(t, c) &= SF_i(t, c) && \text{falls } c \text{ ein Ähnlichkeitsprädikat ist,} \\ eval(t, c_1 \wedge c_2) &= eval(t, c_1) * eval(t, c_2) \\ eval(t, c_1 \vee c_2) &= eval(t, c_1) + eval(t, c_2) - eval(t, c_1 \wedge c_2) \\ eval(t, \neg c) &= 1 - eval(t, c) \end{aligned}$$

Die Auswertung atomarer Ähnlichkeitsprädikate wird mittels sogenannter *Scoring-Funktionen* (SF_i) durchgeführt. Sie ermitteln einen reellen Wert aus dem Intervall $[0; 1]$, der als Relevanzwahrscheinlichkeit bezüglich des jeweiligen Ähnlichkeitsprädikates interpretiert werden kann. Ähnlichkeitsprädikate werden gemäß den verwendeten Auswertungsregeln als unabhängige Ereignisse verstanden. Sie dürfen deshalb innerhalb einer Anfrage nicht mehrfach mit unterschiedlichen Vergleichskonstanten auftreten. Damit wäre z.B. eine Kombination der Ähnlichkeitsprädikate *Ort in der Nähe von Cottbus* und *Ort in der Nähe von Berlin* unzulässig, da diese offensichtlich korrelieren. So könnten sie z.B. nicht gleichzeitig auf 1 (vollständig erfüllt) ausgewertet werden, da es sich um geographisch unterschiedliche Städte handelt.

Des Weiteren wird für die semantisch korrekte Anwendung der obigen Auswertungsfunktionen eine syntaktische Normalisierung der Anfrage notwendig, welche u.a. identische Teilbedingungen zusammenfasst und sich negierende Teilbedingungen eliminiert. Der in [18] vorgeschlagene Normalisierungsalgorithmus basiert auf bekannten, logischen Umformungsregeln, wie z.B. Idempotenz und Distributivität. Diese können hier angewendet werden, da es sich bei der zu Grunde liegenden mathematischen Struktur um eine Boolesche Algebra handelt.

In [14] wird aus diesem rein theoretischen Auswertungsmodell die Kalkülanfragesprache CQQL (Commuting Quantum Query Language) entwickelt. Sie erweitert den relationalen Bereichskalkül um die Behandlung von Ähnlichkeitsprädikaten und Anfragegewichtung. Ein typisches Anwendungsgebiet von CQQL sind Ähnlichkeitsprädikate, welche multimediale Inhalte einbeziehen. Im Kontext des eingeführten Beispielszenarios könnte eine Anfrage folgendermaßen lauten: *Bestimme die Relevanzwahrscheinlichkeit einer Vogelart bezüglich eines Vorgabebildes (VBild), falls sie in der Region Mitteldeutschland ansässig ist.* Die formalisierte CQQL-Anfrage ist gegeben durch:

$$\{(Art, Region, Bild) \mid VArt(Art, Region, Bild) \wedge Region = \text{Mitteldeutschland} \wedge Bild \approx_{BV} VBild\}.$$

Diese Anfrage besitzt mit $(Bild \approx_{BV} VBild)$ ein Ähnlichkeitsprädikat, welches durch eine spezielle Scoring-Funktion für Bildvergleiche (\approx_{BV}) ausgewertet wird.

Allgemein gesprochen wird die Unsicherheit des Anfrageergebnisses auf die Vagheit in der Anfrageformulierung zurückgeführt, wogegen die angefragten Daten selbst als gesichert vorausgesetzt werden: *Eine unsichere Anfrage wird auf einer sicheren Datengrundlage ausgeführt.*

2.2 Many-World-Semantik

Ein weit verbreitetes Semantikmodell für probabilistische Datenbanken ist die Many-World-Semantik [1]. Ausgangspunkt sind eine oder mehrere Tabellen, über welche die Menge aller möglichen Instanzen (hier als Welten oder Zustände bezeichnet) der entsprechenden Relationenschematas betrachtet wird. Die Ausgangstabellen können somit entsprechend ihrer Relationenschemata eine maximal mögliche Menge von Tupeln besitzen. Jede Untermenge dieser maximalen Tupelmenge repräsentiert einen möglichen Zustand der Tabelle. Als Beispiel soll eine Tabelle mit maximal zwei Tupeln betrachtet werden $R(A_1) = \{(1), (2)\}$. Die möglichen vier Zustände lauten hier: $R_{Z_1}(A_1) = \{(1), (2)\}$, $R_{Z_2}(A_1) = \{(1)\}$, $R_{Z_3}(A_1) = \{(2)\}$ und $R_{Z_4}(A_1) = \{\}$. Einer dieser Zustände stellt die Realität dar. Welcher genau dies ist, ist jedoch unbekannt.

Vielmehr wird über Menge der Zustände ein Wahrscheinlichkeitsmaß definiert. Es drückt aus, mit welcher Wahrscheinlichkeit $Pr(Z_i)$ ein bestimmter Zustand Z_i der reale Zustand ist. Zustände können hierbei auch eine Wahrscheinlichkeit von Null besitzen.

Die Wahrscheinlichkeiten der einzelnen Zustände werden anhand der Tupel, welche in dem jeweiligen Zustand existieren definiert. Hierfür ist jedem Tupel t_i eine Eintrittswahrscheinlichkeit $Pr(t_i)$ zugeordnet, die ausdrückt, mit welcher Wahrscheinlichkeit es in der Realität vorkommt.

Prinzipiell ist die Many-World-Semantik nicht auf eine bestimmte Klasse von Wahrscheinlichkeitsmaßen festgelegt. Um jedoch eine möglichst einfache Berechnung der Zustandswahrscheinlichkeiten $Pr(Z_i)$ zu gewährleisten, werden die Eintrittswahrscheinlichkeiten der Tupel $Pr(t_i)$ als untereinander unabhängig angenommen. Dies bedeutet, die Eintrittswahrscheinlichkeit eines bestimmten Tupels ändert sich nicht mit dem Vorhandensein oder dem Nicht-Vorhandensein eines beliebigen anderen Tupels. Somit ergibt sich die Wahrscheinlichkeit eines Zustandes als $Pr(Z_i) = \prod_{t_i \in Z_i} (Pr(t_i)) * \prod_{t_i \notin Z_i} (1 - Pr(t_i))$.

Mit Eintrittswahrscheinlichkeiten für Tupel lassen sich u.a. besonders gut Beobachtungen und Bewertungen modellieren, welche einer bestimmten Unsicherheit bzw. Subjektivität unterliegen. Die Eintrittswahrscheinlichkeiten/Konfidenzwerte solcher Beobachtungen bzw. Bewertungen werden meist durch Expertenwissen bestimmt, das sich meist nur sehr unzureichend in Funktionen oder automatischen Verfahren abbilden lässt.

In dem eingeführten Beispielszenario stellen die Tupel der Tabelle $VBeob$ solche subjektiven Beobachtungen dar. Die im Attribut Pr hinterlegten Eintrittswahrscheinlichkeiten sind von dem jeweiligen Beobachter auf Basis seines eigenen individuellen Erfahrungshorizonts bestimmt worden.

Eintrittswahrscheinlichkeiten von Tupeln aus einer Datenrelation stellen singuläre Basisereignisse dar. Dem gegenüber stehen *komplexe* Ereignisse, welche im Zuge der Anfrageauswertung aus der Kombination von Basisereignissen konstruiert werden.

Eine typische Many-World-Anfrage mit komplexen Ereignissen könnte wie folgt lauten: *Bestimme alle Zweier-Kombinationen von unterschiedlichen Vogelarten, die am selben Ort beobachtet worden sind.* Wenn man die Beispielanfrage

auf die Tabelle $VBeob$ angewendet ergibt sich u.a. die Kombination Star und Amsel. Das Eintreten dieser Kombination stellt ein komplexes Ereignis dar, welches sich aus zwei gleichzeitig eintretenden unabhängigen Basisereignissen zusammensetzt: $Pr((Star, Amsel, Cottbus)) = Pr((Star, Cottbus, September)) * Pr((Amsel, Cottbus, Mai)) = 0.36$

Zusammenfassend kann festgestellt werden, dass im Gegensatz zum vorherigen Semantikmodell hier die Daten als unsicher betrachtet werden: *Eine sichere Anfrage wird auf einer unsicheren Datengrundlage ausgeführt.*

2.3 Die Erweiterung der Many-World-Semantik um Relevanzwahrscheinlichkeiten

Die Kombination der beiden oben beschriebenen Semantikmodelle ergibt eine erweiterte Klasse von Anfragen. Ausgehend von einem Tupel in einer bestimmten Welt kann nun zusätzlich die Relevanz dieses Tupels bezüglich einer Ähnlichkeitsanfrage betrachtet werden. Als Beispiel soll folgende Anfrage gestellt werden: *Bestimme alle Vogelarten, welche beobachtet worden sind und zusätzlich möglichst ähnlich einem Vorgabebild ($VBild$) sind.* Die Bedingung kann wie folgt formalisiert werden:

$$VBeob(Art, Ort, Zeit) \wedge VArt(Art, Region, Bild) \wedge Bild \approx_{BV} VBild.$$

In dieser Beispielanfrage wird die Eintrittswahrscheinlichkeit der Beobachtung mit der Relevanzwahrscheinlichkeit der Beobachtung bezüglich des Ähnlichkeitsprädikates $Bild \approx_{BV} VBild$ verknüpft.

Die Kombination beider Anfrageparadigmen wird immer dann interessant, wenn konstruierte Datenobjekte mit komplexen Eintrittsereignissen assoziiert werden und auf den Attributwerten dieser Datenobjekte logikbasierte Ähnlichkeitsanfragen ausgeführt werden. Es wird somit eine Verbindung zwischen einer *subjektiven* Quantifizierung von Ereignissen und der *objektiven* Berechnung von Ähnlichkeitswerten realisiert.

In Anlehnung an die beiden vorangegangenen Abschnitte kann folgender Grundsatz für die Kombination von Relevanzwahrscheinlichkeiten und Many-World-Semantik formuliert werden: *Eine unsichere Anfrage wird auf einer unsicheren Datengrundlage ausgeführt.*

3 CQQL^P - Die probabilistische Erweiterung der Anfragesprache CQQL

Im vorherigen Kapitel wurde die erweiterte Anfrageklasse vorgestellt, welche sich aus der Kombination von Relevanzwahrscheinlichkeiten und der Many-World-Semantik ergibt.

Die technische Berechnung der kombinierten Wahrscheinlichkeiten basiert auf einem integrierten Wahrscheinlichkeitsmaß, welches auf einem Produktwahrscheinlichkeitsraum zwischen der Menge aller möglichen Welten und der Menge aller Anfrageräume definiert wird [11].

Die daraus resultierende probabilistische Erweiterung von CQQL wird als $CQQL^P$ bezeichnet. In den folgenden Abschnitten werden grundlegende Konzepte von $CQQL^P$ vorgestellt. Eine genaue Definition von $CQQL^P$ wird in [9] gegeben.

3.1 Probabilistische Relationen und probabilistische Relationenprädikate

Als erster Schritt wird das Konzept der *probabilistischen Relation* in den Sprachumfang von $CQQL^P$ eingeführt. In probabilistischen Relationen besitzt jedes Tupel eine individuelle Eintrittswahrscheinlichkeit. Die Eintrittswahrscheinlichkeit stellt dabei kein explizites Attribut dar, d.h. sie kann nicht direkt manipuliert werden. Die definierten Eintrittswahrscheinlichkeiten werden als untereinander unabhängig vereinbart.

Bisher konnte eine CQQL-Formel aus drei verschiedenen Typen von Prädikaten bestehen [14]: (1) Relationenprädikate (z.B. $R_1(X_1, X_2)$), (2) Boolesche Prädikate (z.B. $X_1 = 2$ oder $X_2 < 5$) und (3) Ähnlichkeitsprädikate (z.B. $X_3 \approx 4$). Für die Auswertung von probabilistischen Relationen wird in $CQQL^P$ der neue Typ der *probabilistischen Relationenprädikate* (Notation: $R_i^{\approx}(X_1, \dots, X_n)$) eingeführt. Wird ein solches probabilistisches Relationenprädikat auf ein bestimmtes Tupel angewendet, ist der entsprechende Rückgabewert die Eintrittswahrscheinlichkeit dieses Tupels, falls es sich in der Relation befindet. Andernfalls wird der Wert 0 zurück gegeben.

Als Anwendungsbeispiel wird folgende Anfrage betrachtet: *Bestimme alle Vogelarten, welche in Cottbus im September beobachtet worden sind.* Die formalisierte Anfrage in $CQQL^P$ lautet:

$$\{(Art, Ort, Zeit) \mid VBeob^{\approx}(Art, Ort, Zeit) \wedge Ort = Cottbus \wedge \\ Zeit = September\}.$$

Die Auswertung der Anfrage ergibt für das Tupel (Star, Cottbus, September) der Relation $VBeob$ eine Wahrscheinlichkeit von $eval(VBeob^{\approx}(Star, Cottbus, September)) * eval(Cottbus = Cottbus) * eval(September = September) = 0.9 * 1 * 1 = 0.9$ und für das Tupel (Fink, Berlin, Juni) von $eval(VBeob^{\approx}(Fink, Berlin, Juni)) * eval(Berlin = Cottbus) * eval(Juni = September) = 0.5 * 0 * 0 = 0$.

3.2 Probabilistische Normalisierung

Ein zentraler Bestandteil der CQQL-Auswertung ist die syntaktische Normalisierung von Anfragen. Sie garantiert die semantisch korrekte Aggregation der Relevanzwahrscheinlichkeiten von Ähnlichkeitsprädikaten. So wird etwa die Beispielbedingung $(Ort \approx_{OV} Cottbus) \wedge (Ort \approx_{OV} Cottbus)$ zu $(Ort \approx_{OV} Cottbus)$ normalisiert, weil es sich semantisch um die Konjunktion ein und derselben Bedingung handelt (\approx_{OV} ist Ähnlichkeitsoperator für ein Ortsvergleich). Die direkte Auswertung der unnormalisierten Anfrage würde eine falsche Relevanzwahrscheinlichkeit von $eval(Ort \approx_{OV} Cottbus) * eval(Ort \approx_{OV} Cottbus)$ anstatt von $eval(Ort \approx_{OV} Cottbus)$ ergeben.

Die Normalisierung von Ähnlichkeitsprädikaten wird nun auf probabilistische Relationenprädikate übertragen. Dadurch wird z.B. gewährleistet, dass Eintrittswahrscheinlichkeiten gleicher Tupel nicht mehrfach in die Gesamtwahrscheinlichkeit eingehen. Als Beispiel wird der Schnitt der Relation $VBeob$ mit sich selbst betrachtet: $\{(Art, Ort, Zeit) \mid VBeob \approx (Art, Ort, Zeit) \wedge VBeob \approx (Art, Ort, Zeit)\}$. Sobald man ein konkretes Tupel mit Hilfe dieser unnormalisierten Bedingung auswertet, erkennt man, dass die Eintrittswahrscheinlichkeiten ein und desselben Tupels zweimal in die Gesamtwahrscheinlichkeit des Ergebnistupels eingehen würde. Dies widerspricht der probabilistischen Many-World-Semantik. Auch hier ist eine Normalisierung der Formel notwendig. In diesem Fall vereinfacht sich die Bedingung zu $VBeob \approx (Art, Ort, Zeit)$.

3.3 Intra-Tupel versus Inter-Normalisierung

Im letzten Abschnitt wurde zum einen die Normalisierung von Ähnlichkeitsprädikaten und zum anderen die Normalisierung von probabilistischen Relationenprädikaten vorgestellt. Die erste Normalisierung garantiert die korrekte Aggregation von Relevanzwahrscheinlichkeiten, die zweite ist dagegen dafür verantwortlich, dass Eintrittswahrscheinlichkeiten semantisch richtig zusammengefasst werden.

Betrachtet man die Normalisierung von Ähnlichkeitsprädikaten genauer, erkennt man, dass sich die zu normalisierenden Ereignisse auf Attributwerte genau eines Tupels bzw. genau einer Variablenbelegung beziehen. Dies entspricht exakt dem quantenlogischen Auswertungsmodell, da hier die Auswertung für einen einzelnen Vektor gegenüber einem Anfrageraum definiert wird. Eine Interaktion zwischen verschiedenen Vektoren innerhalb der Auswertung ist nicht vorgesehen. Daher kann die Normalisierung von Ähnlichkeitsprädikaten als *Intra-Tupel*-Normalisierung bezeichnet werden. Sie wirkt nur innerhalb eines Tupels bzw. einer Variablenbelegung.

Die Normalisierung von probabilistischen Relationenprädikaten unterstützt dagegen die Bildung von komplexen Ereignissen, welche die Eintrittswahrscheinlichkeiten von konstruierten Tupeln verkörpern. Komplexe Ereignisse dieser Art beziehen sich definitionsgemäß auf mehrere Basistupel bzw. Variablenbelegungen. Demnach findet eine *Inter*-Normalisierung zwischen mehreren Tupeln bzw. Variablenbelegungen statt. Eine typische Operation, die eine Inter-Normalisierung notwendig macht, ist die Projektion. Hier können mehrere Ausgangstupel zu einem Ergebnistupel verdichtet werden. Die Wahrscheinlichkeit des Ergebnistupel ergibt sich aus einer disjunktiven Verknüpfung der Wahrscheinlichkeiten der jeweiligen Ausgangstupel. Dies bedeutet, dass mindestens eines der Ausgangsereignisse eingetreten sein muss, um das Ereignis des verdichteten Tupels zu erzeugen [6].

Die durch die Projektion erzeugte Disjunktion muss jedoch mit einer Inter-Normalisierung behandelt werden, da Basisereignisse mehrfach in den möglicherweise komplexen Ereignissen der Ausgangstupel vorliegen können.

Als Beispiel soll die folgende Anfrage betrachtet werden: *Bestimme alle Vogelarten, welche in der Nähe von Berlin oder in der Nähe von Berlin beobachtet worden sind.* Die formalisierte Variante dieser Anfrage lautet:

$$\{(Art) \mid \exists Ort : \exists Zeit : VBeob \approx (Art, Ort, Zeit) \wedge (Ort \approx_{OV} Berlin \vee Ort \approx_{OV} Berlin)\}.$$

Das doppelte Auftreten eines Ähnlichkeitsprädikates kann z.B. durch die automatisierte Generierung von Anfragen oder durch die Anwendung von Sichten auftreten.

Bei der Auswertung der Beispielanfrage muss sowohl eine Intra-Tupel- als auch eine Inter-Normalisierung durchgeführt werden. Zunächst wird die Intra-Tupel-Normalisierung auf die Bedingung $(Ort \approx_{OV} Berlin \vee Ort \approx_{OV} Berlin)$ angewendet: $(Ort \approx_{OV} Berlin)$. Somit ergeben sich im ersten Schritt für das Tupel (Star, Cottbus, September) die Wahrscheinlichkeit $eval(VBeob \approx (Star, Cottbus, September) \wedge Cottbus \approx_{OV} Berlin)$ und für das Tupel (Star, Cottbus, August) die Wahrscheinlichkeit $eval(VBeob \approx (Star, Cottbus, August) \wedge Cottbus \approx_{OV} Berlin)$. Anschließend muss eine Inter-Normalisierung auf die Projektion¹ des Attributes *Art* durchgeführt werden. Da sich die Inter-Normalisierung nur auf probabilistische Relationenprädikate bezieht, verändert sie die disjunktiv konstruierte Formel für das Ergebnistupel (Star) hier nicht mehr:

$$\begin{aligned} & eval((VBeob \approx (Star, Cottbus, September) \wedge Cottbus \approx_{OV} \\ & Berlin) \vee (VBeob \approx (Star, Cottbus, August) \wedge Cottbus \approx_{OV} Berlin)) = \\ & (1 - (1 - (0.9 * 0.7))(1 - (0.3 * 0.7))) = 0.7077, \end{aligned}$$

wenn $eval(Cottbus \approx_{OV} Berlin)$ als 0.7 angenommen wird².

4 Die probabilistische Anfragesprache QSQL^p

Die Anfragesprache SQL ist der etablierte Standard für den Zugriff auf objektrelationale Datenbanksysteme. Seit der Einführung von SQL in den 70er Jahren ist ihre praktische Relevanz kontinuierlich gestiegen. Aus diesem Grund werden die in Kapitel 3 vorgestellten Konzepte der Kalkülanfragesprache CQQL^p auf SQL übertragen. Dadurch werden sie in Form des SQL-Dialektes QSQL^p einer breiten Entwicklerschicht zugänglich gemacht. Der bisherige Funktionsumfang von SQL bleibt dabei vollständig in QSQL^p erhalten, d.h. alle SQL-Anfragen können auch in QSQL^p wie gewohnt formuliert und ausgewertet werden.

Tupel von probabilistischen Relationen besitzen eine individuelle Eintrittswahrscheinlichkeit. QSQL^p benutzt als Eintrittswahrscheinlichkeit automatisch die Werte des Attributes *probvalue*, falls es in der Relation vorhanden ist. Andernfalls wird für jedes Tupel implizit eine Eintrittswahrscheinlichkeit von 1 angenommen. Neben der expliziten Speicherung der Eintrittswahrscheinlichkeiten

¹ Im Kalkül wird eine Projektion mittels (mehrerer) Existenzquantoren ausgedrückt, welche die nicht projizierten Attribute binden.

² Wegen der DeMorgan-Umformungsregel gilt: $eval(A \vee B) = eval(\neg(\neg A \wedge \neg B)) = (1 - (1 - eval(A)) * (1 - eval(B)))$

können diese auch mittels von Unterabfragen berechnet werden. Die berechneten Wahrscheinlichkeiten befinden sich dann wiederum in dem Attribut *probvalue* der Ergebnisrelation.

Die Selektion von Tupeln aus einer oder mehreren Tabellen wird syntaktisch wie in SQL formuliert. So wird die logische Anfragebedingung, welche sich aus Booleschen Prädikaten, Ähnlichkeitsprädikaten, sowie den logischen Operatoren *and*, *or* und *not* zusammensetzt, ebenfalls in der *where*-Klausel einer Anfrage platziert. Gegenüber SQL können in QSQL^P zusätzlich Ähnlichkeitsbedingungen mittels des Ähnlichkeitsoperators \approx formuliert werden. Eine Beispielanfrage in QSQL^P wird in Abschnitt 4.3 vorgestellt.

4.1 Der Auswertungsprozess von QSQL^P

Die interne Ergebnisberechnung einer QSQL^P-Anfrage wird mittels einer Transformation zwischen den folgenden drei Anfragesprachen realisiert: (1) QSQL^P zur Formulierung der Anfrage, (2) die Ähnlichkeitsalgebra QA^P zur Normalisierung und Optimierung, sowie (3) SQL-99 zur eigentlichen Berechnung des Ergebnisses innerhalb eines DBMS (siehe Abbildung 3).

In den nächsten Abschnitten wird die Normalisierung und Optimierung von QA^P-Ausdrücken skizziert. Eine exakte Definition der Ähnlichkeitsalgebra QA^P, sowie eine detaillierte Beschreibung der Abbildung von QSQL^P nach QA^P wird in [12] gegeben. Die verwendeten Prinzipien für die finale Abbildung nach SQL-99 wurden bereits in der bisherigen QSQL-Version eingesetzt und werden in [13] vorgestellt.

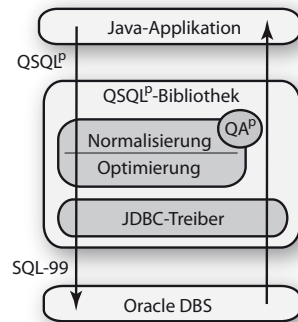


Abb. 3. Auswertungsprozess

4.2 Die Ähnlichkeitsalgebra QA^P

Das Kernstück der Auswertung von QSQL^P-Anfragen ist die Erzeugung von semantisch äquivalenten Ausdrücken in QA^P und deren Optimierung.

Die probabilistische Normalisierung von Prädikaten wurde bereits im Kontext von CQQL^P in Kapitel 3 diskutiert. Die dort entwickelten Konzepte werden nun auf die Ähnlichkeitsalgebra QA^P angewendet. Damit werden die Ähnlichkeitsalgebra QA^P und die Menge der *sicheren* CQQL^P-Anfragen gleichmächtig [12]. Eine CQQL^P-Anfrage gilt als sicher, wenn ihre Ergebnismenge endlich ist und sie darüber hinaus in endlicher Zeit berechnet werden kann.

Die Operatoren der Ähnlichkeitsalgebra QA^P werden in Tabelle 1 aufgeführt. Das Ergebnis eines jeden Operators ist ein Tupel (R, Pr) , welches aus dem relationalen Datenanteil R und der Wahrscheinlichkeitsfunktion Pr besteht. Die

Funktion Pr ordnet jedem Tupel aus R eine Wahrscheinlichkeit zu. Die Berechnung von R wird dabei mit den bekannten Operatoren aus der Relationalen Algebra durchgeführt.

Probabilistische Auswertungsoperatoren werden gemeinhin in *extensionale* und *intensionale* Operatoren unterteilt (siehe z.B. [4], [6]). Dabei aggregieren extensionale Operatoren Wahrscheinlichkeiten ohne die zu Grunde liegenden (komplexen) Ereignisse zu berücksichtigen. Die richtige Semantik muss vielmehr durch die richtige Anordnung der Operatoren innerhalb des Ausdrucks garantiert werden. Dagegen besitzen intensionale Operatoren zur Berechnung der richtigen Ergebniswahrscheinlichkeiten eine interne Normalisierung. Diese stellt im Allgemeinen einen signifikanten Mehraufwand dar.

Operation	Semantik
<u>(Prob.) Relation R, R^p</u>	$R := R$ $Pr(t) := 1$ für R bzw. $Pr(t)$ wird gesetzt für R^p
<u>Projektion -extens.-</u> $\pi_{\mathcal{A}}^e(E_1)$	$R := \pi_{\mathcal{A}}^{RA}(R_1)$ $Pr(t) := 1 - \prod_{\tilde{t} \in \{\tilde{t} \in R_1 \mid \tilde{t}[\mathcal{A}] = t\}} (1 - Pr(\tilde{t}))$
<u>Projektion -intens.-</u> $\pi_{(\mathcal{A}, F)}^i(E_1)$	$R := \pi_{\mathcal{A}}^{RA}(R_1)$ $Pr(t) := eval(norm_{inter}(\bigvee_{\tilde{t} \in \{\tilde{t} \in R_1 \mid \tilde{t}[\mathcal{A}] = t\}} F(\tilde{t})))$
<u>Selektion</u> $\sigma_F(E_1)$	$R := \{t \in R_1 \mid Pr(t) > 0\}$ $Pr(t) := Pr_1(t) * eval(norm_{intra}(F(t)))$
<u>Schnitt</u> $E_1 \cap_{(\mathcal{A}_1, \mathcal{A}_2)} E_2$	$R := R_1 \bowtie_{natural}^{RA} \beta_{(\mathcal{A}_1 \leftarrow \mathcal{A}_2)}(R_2)$ $Pr(t) := Pr_1(t[R_1]) * Pr_2(t[R_2])$
<u>Vereinigung</u> $E_1 \cup_{(\mathcal{A}_1, \mathcal{A}_2)} E_2$	$R := R_1 \bowtie_{full}^{RA} outer \beta_{(\mathcal{A}_1 \leftarrow \mathcal{A}_2)}(R_2)$ $Pr(t) := \begin{cases} Pr_1(t[R_1]) + Pr_2(t[R_2]) - & \text{falls } t[R_1] \in R_1 \wedge \\ Pr_1(t[R_1]) * Pr_2(t[R_2]) & t[R_2] \in R_2 \\ Pr_1(t[R_1]) & \text{falls } t[R_1] \in R_1 \wedge \\ & t[R_2] \notin R_2 \\ Pr_2(t[R_2]) & \text{falls } t[R_1] \notin R_1 \wedge \\ & t[R_2] \in R_2 \end{cases}$
<u>Differenz</u> $E_1 -_{(\mathcal{A}_1, \mathcal{A}_2)} E_2$	$R := R_1 \bowtie_{left}^{RA} outer \beta_{(\mathcal{A}_1 \leftarrow \mathcal{A}_2)}(R_2)$ $Pr(t) := \begin{cases} Pr_1(t[R_1]) * & \text{falls } t[R_1] \in R_1 \wedge \\ (1 - Pr_2(t[R_2])) & t[R_2] \in R_2 \\ Pr_1(t[R_1]) & \text{falls } t[R_1] \in R_1 \wedge \\ & t[R_2] \notin R_2 \end{cases}$
<u>Kreuzprodukt</u> $E_1 \times E_2$	$R := R_1 \times^{RA} R_2$ $Pr(t) := Pr_1(t[R_1]) * Pr_2(t[R_2])$

Tabelle 1. Übersicht der QA^p-Operatoren

4.3 Die Abbildung von QSQL^P nach QA^P

Da die Auswertung einer QSQL^P-Anfrage mittels QA^P-Ausdrücke geschieht, ist die Semantik von QSQL^P mittels der Abbildung von QSQL^P nach QA^P und der Definition der QA^P-Operatoren festgelegt. Dies wiederum bedingt die Gleichmächtigkeit zwischen der *Kernfunktionalität* von QSQL^P und dem sicheren CQQL^P-Kalkül, da bereits eine Äquivalenz zwischen QA^P und CQQL^P festgestellt wurde. Der Begriff Kernfunktionalität bezieht sich auf den Umstand, dass bestimmte SQL-Funktionalitäten wie die Gruppierung und die Multimengen-Semantik nicht direkt in eine Kalkülsprache, welche auf Prädikatenlogik 1. Stufe basiert, übertragen werden können.

Der Dreiklang von sicherem CQQL^P (Kalkül), QA^P (Algebra) und QSQL^P (SQL) spielt bei der Abbildung von QSQL^P nach QA^P eine wesentliche Rolle.

Der Ausgangspunkt für die folgenden Betrachtung ist eine in QSQL^P formulierte Anfrage. Als Grundlage für die Erzeugung eines entsprechenden gleichwertigen QA^P-Ausdrucks wird die Kalkülauswertung einer äquivalenten CQQL^P-Anfrage betrachtet.

In der Kalkülauswertung wird jede Variablenbelegung gegen eine normalisierte Bedingung F ausgewertet. Die Menge aller gebundenen Variablenbelegungen wird hier als R_{VB} bezeichnet. Die eigentlichen Ergebnistupel werden abschließend anhand einer Menge von Ausgabeattributen \mathcal{A} gebildet. Übersetzt man dieses Vorgehen direkt in einen Algebraausdruck ergibt sich folgende Grundstruktur für die Auswertung: $\pi_{\mathcal{A}}(\sigma_F(R_{VB}))$.

In dem grundlegenden Algebraausdruck wird die Menge R_{VB} als Eingangsrelation benutzt. Offensichtlich kann diese Relation schnell anwachsen, da sie alle benötigten Variablenbelegungen als Tupel beinhaltet und Projektionen bzw. Selektionen, welche die Eingangsrelation verkleinern würden, erst abschließend durchgeführt werden. Eine direkte Auswertung dieses Ausdrucks ist demnach nicht praktikabel. Bevor im nächsten Abschnitt auf eine notwendige Optimierung eingegangen wird, steht hier zunächst die Generierung der Grundstruktur $\pi_{\mathcal{A}}(\sigma_F(R_{VB}))$ im Vordergrund.

Die übergebene QSQL^P-Anfrage wird hierfür in eine spezielle Datenstruktur, dem sogenannten *Select-From-Where-Baum*, überführt. Er stellt die Grundlage für den Abbildungsalgorithmus zwischen QSQL^P und QA^P dar. Im SFW-Baum wird u.a. die syntaktische Struktur der QSQL^P-Anfrage nachgebildet. Dementsprechend sind die Knoten des Baumes entweder SFW-Blöcke, Relationen oder Relationsoperatoren ($\times, \cup, \cap, -$). Jeder SFW-Block besitzt (1) eine *Projektionsliste*, welche aus der *select*-Klausel generiert wird, (2) eine *logische Bedingung*, welche auf der *where*-Klausel basiert, und (3) Konnektoren zu weiteren möglichen Unterabfragen.

Als Beispiel wird die abstrakte QSQL^P-Anfrage aus Quelltext 4.1 betrachtet. Die Anfrage drückt den Schnitt zweier probabilistischer Tabellen T_1 und T_2 aus, wobei die bereinigten Relationenschemata (ohne Attribut Pr) der benutzten Tabellen $R(T_1) = (A_1, A_2, A_3)$ und $R(T_2) = (B_1, B_2)$ lauten. Die Anfrage beinhaltet u.a. die zwei Ähnlichkeitsbedingungen $B_1 \approx 1$ und $A_1 \approx 1$. Diese be-

```

select A1
from
  ( select A1, A2
    from T1
    where A3 ~ 3 and A2 > 2 )
intersect
  ( select *
    from T2
    where B1 ~ 1 and B2 ~ 2 )
where A1 ~ 1

```

Quelltext 4.1. Beispielanfrage in QSQL^p

ziehen sich auf ein und dasselbe Attribut, wenn man die geschnittene Relation als Grundlage betrachtet. Diese Überlappung von Ähnlichkeitsprädikaten muss mittels einer Intra-Tupel-Normalisierung aufgelöst werden. Andernfalls wird auf den ersten Attributwert eines jeden Tupels aus T_2 die Bedingung *ähnlich 1* doppelt ausgeführt.

Der für die Beispielanfrage generierte SFW-Baum wird in Abbildung 4 gezeigt.

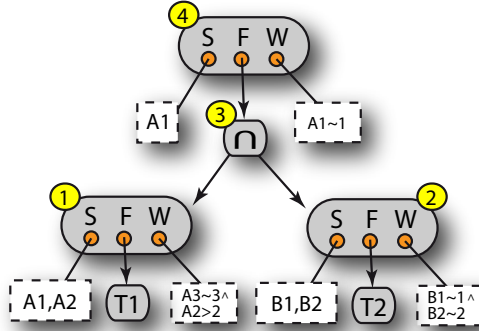


Abb. 4. SFW-Baum der Anfrage aus Quelltext 4.1

Mit Hilfe einer Traversierung des SFW-Baum werden knotenweise die drei Bestandteile der initialen Grundstruktur $\pi_{\mathcal{A}}(\sigma_F(R_{VB}))$ konstruiert, d.h. (1) die Attributmenge \mathcal{A} , (2) die Selektionsbedingung F und (3) der Algebraausdruck zur Konstruktion von R_{VB} .

Die Tabelle 2 beinhaltet die drei QA^p-Bestandteile \mathcal{A} , F und R_{VB} für die Knoten 1 bis 4 der Beispielanfrage. Die Formeln der beiden Knoten 1 und 2 ergeben sich zu F_1 und F_2 . Die Bereichsvariablen X_i stammen aus einem globalen

	\mathcal{A}	F	R_{VB}
1	$\mathcal{A}_1 = \{X_1, X_2\}$	$F_1 = T_1^\approx(X_1, X_2, X_3) \wedge X_3 \approx 3 \wedge X_2 > 2$	T_1
2	$\mathcal{A}_2 = \{X_4, X_5\}$	$F_2 = T_2^\approx(X_4, X_5) \wedge X_4 \approx 1 \wedge X_5 \approx 2$	T_2
3	$\mathcal{A}_3 = \{X_1, X_2\}$	$F_3 = (T_1^\approx(X_1, X_2, X_3) \wedge X_3 \approx 3 \wedge X_2 > 2) \wedge (T_2^\approx(X_1, X_2) \wedge X_1 \approx 1 \wedge X_2 \approx 2)$	$T_1 \cap_{(\mathcal{A}_1, \mathcal{A}_2)} T_2$
4	$\mathcal{A}_4 = \{X_1\}$	$F_4 = ((T_1^\approx(X_1, X_2, X_3) \wedge X_3 \approx 3 \wedge X_2 > 2) \wedge (T_2^\approx(X_1, X_2) \wedge X_1 \approx 1 \wedge X_2 \approx 2)) \wedge X_1 \approx 1$	$T_1 \cap_{(\mathcal{A}_1, \mathcal{A}_2)} T_2$

Tabelle 2. Berechnung des initialen Grundausrdruckes

Variablenschemata und repräsentieren die jeweiligen Attribute der zu Grunde liegenden Relationen T_1 und T_2 . Die Relationen T_1 und T_2 wiederum erzeugen die probabilistischen Relationenprädikate T_1^\approx und T_2^\approx . Sie werden genutzt um die entsprechenden Eintrittswahrscheinlichkeiten einfließen zu lassen. Logische Bedingungen aus der *where*-Klausel werden konjunktiv an die jeweiligen probabilistischen Relationenprädikate gebunden.

Die beiden Zwischenformeln F_1 und F_2 werden in Knoten 3 zu der Formel F_3 kombiniert. Der Schnittoperator kann dabei direkt in eine Konjunktion zwischen F_1 und F_2 umgewandelt werden, wobei die beiden Variablenschemata einander angepasst werden müssen. Dadurch können äußere Bedingungen (hier: $A_1 \approx 1$) auf die Tupel beider Eingangsrelationen wirken.

Die Attributmengen \mathcal{A}_i ergeben sich direkt aus den Projektionsattributlisten der entsprechenden SFW-Blöcke. Der Algebraausdruck zur Berechnung von R_{VB} wird entsprechend den Abbildungsvorschriften aus [12] generiert.

Der initiale QA^p -Ausdruck ergibt sich dann zu $\pi_{(\{X_1\}, F)}^i(T_1 \cap_{(\mathcal{A}_1, \mathcal{A}_2)} T_2)$, wobei $F = \text{norm}_{\text{intra}}(F_4) = T_1^\approx(X_1, X_2, X_3) \wedge T_2^\approx(X_1, X_2) \wedge X_1 \approx 1 \wedge X_2 \approx 2 \wedge X_2 > 2 \wedge X_3 \approx 3$. In F ist nun die Überlappung der Ähnlichkeitsprädikate aufgelöst, da $B_1 \approx 1$ und $A_1 \approx 1$ jeweils auf $X_1 \approx 1$ abgebildet und mittels der Idempotenz-Regel zusammengefasst worden sind.

4.4 Optimierung in QA^p

Um ein starkes Anwachsen von R_{VB} zu vermeiden, muss der initiale Grundausrdruck optimiert werden. Die Optimierung von QA^p -Ausdrücken setzt die Möglichkeit einer separaten Normalisierung von Teilausdrücken voraus. Dies bedeutet, dass zwischen zu trennenden Teilausdrücken keine Überlappungen von Ähnlichkeitsprädikaten existieren dürfen, die aufgelöst werden müssten.

Ein optimierter QA^p -Ausdruck kann extensionale, sowie intensionale Operatoren beinhalten. Ziel der Optimierung ist es einen Ausdruck zu erzeugen der möglichst auf die Anwendung von intensionalen Operatoren verzichtet, da diese einen internen Normalisierungsschritt (siehe Tabelle 1) notwendig machen. Zur Verdeutlichung des Optimierungspotential, soll ein optimierter Ausdruck für das eingeführte Beispiel in Abbildung 5 genutzt werden. Der optimierte Ausdruck enthält nur noch extensionale Operatoren. Die Normalisierung der Ausgangs-

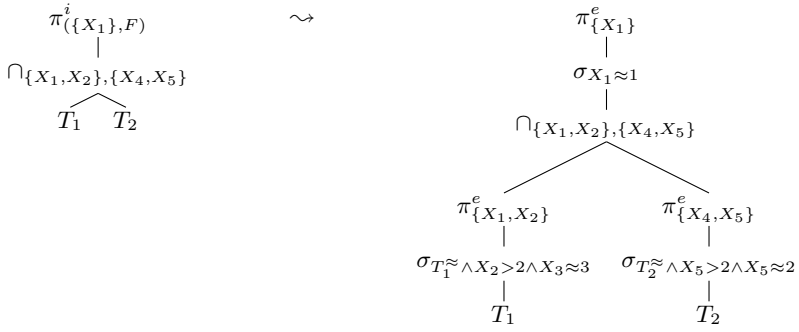


Abb. 5. Optimierung des initialen Algebrabaumes

anfrage verschiebt sich auf die gezielte Anwendung extensionaler Algebraoperatoren und den Einsatz entsprechender Selektionsbedingungen. Die konzeptionelle Konstruktion von R_{VB} vereinfacht sich durch den Einsatz extensionaler Projektionen zu einer einfachen Schnittoperation, wenn man den relationalen Datenanteil des Operators $\cap_{\{X_1, X_2\}, \{X_4, X_5\}}$ (siehe Tab. 1) als natürlichen Verbund zwischen zwei Relationen mit gleichen Relationenschemata auflöst. Damit gleicht der erzeugte Ausdruck stark der ursprünglichen QSQL^p-Anfrage. Der gewonnene Effekt neben der Wahrscheinlichkeitsberechnung ist die Normalisierung der überlappenden Ähnlichkeitsprädikate $B_1 \approx 1$ und $A_1 \approx 1$, welche beide auf $X_1 \approx 1$ abgebildet worden sind.

5 Experimente

Zur Evaluierung der Performanz wurde das Beispiel aus Quelltext 4.1 mit den unoptimierten und optimierten Ausführungsplänen aus Abbildung 5 untersucht. Zu Grunde lagen zwei Familien von Tabellen T_1, T_2 , welche jeweils $10^0, 10^1, \dots, 10^6$ Tupel enthielten. Zur Überprüfung wurde ein Sun UltraSPARC IV 1.4 GHz mit 8 GB RAM genutzt. Bei Experiment 1 enthielt Tabelle T_1 konstant 10^4 Tupel. Wie in Abbildung Tabelle 3 zu erkennen ist, wächst die Laufzeit der nicht optimierten Anfrage linear mit der Größe von T_2 , während die optimierte Anfrage deutlich weniger Zeit benötigt.

Bei Experiment 2 wuchsen beide Tabellen T_1 und T_2 . In Tabelle 4 sieht man, dass die Laufzeit des optimierten Verfahrens in diesem Fall linear wächst, während die Laufzeit des nicht optimierten Verfahrens quadratisch wächst.

Das nicht optimierte Verfahren ist zwar semantisch korrekt, aber zu langsam. Das äquivalente optimierte Verfahren ist also trotz seiner benötigten komplexeren Konstruktion bei Anfragen auf große Tabellen zu bevorzugen.

6 Vergleichbare Ansätze

In der Literatur wurden eine Vielzahl von Systemen vorgeschlagen, welche die probabilistische Verarbeitung von relationalen Daten unterstützen. In dem Kon-

Anfragen	Anzahl Tupel in T_2						
	10^0	10^1	10^2	10^3	10^4	10^5	10^6
optimiert	0,5	0,5	1,5	1,5	2,0	9,0	82,3
nicht optimiert	0,5	0,6	3,5	30,1	297,3	-	-

Tabelle 3. Auswertungszeit in Sekunden bei 10^4 Tupel in T_1

Anfragen	Anzahl Tupel in $T_1 =$ Anzahl Tupel in T_2						
	10^0	10^1	10^2	10^3	10^4	10^5	10^6
optimiert	0,5	0,5	0,5	0,5	2,0	16,8	162,7
nicht optimiert	0,5	0,5	0,5	3,5	297,3	-	-

Tabelle 4. Auswertungszeit in Sekunden

text von $QSQL^P$ sollen vor allem Ansätze untersucht werden, die eine *logikbasierte Anfragesprache* in Form eines Kalküls, einer Algebra oder eines SQL-Dialektes anbieten.

Die betrachteten Systeme können bezüglich der Wahrscheinlichkeitsberechnung grob in zwei Klassen eingeteilt werden: extensionale und intensionale Ansätze. Die konzeptionellen Charakteristika von extensionalen und intensionalen Verfahren werden in [16] umfassend diskutiert.

Extensionale Systeme [3,2,5,4] können sehr effizient Wahrscheinlichkeiten berechnen, wenn die unterstützte Klasse von Anfragen oder die Klasse der verwendeten Wahrscheinlichkeitsmaße eingeschränkt wird.

Zum Beispiel nehmen Cavallo und Pittarelli in [3] an, dass Tupel in derselben Relation disjunkte Ereignisse darstellen. Barbara et. al. [2] verallgemeinern dieses Modell, sodass Tupel unabhängig und deren Attribute zusätzlich ungenau sein können, was zu disjunkten Eintrittswahrscheinlichkeiten auf Attributebene führt. Dabei muss jede Relation eine Menge von deterministischen Attributen besitzen, welche den Schlüssel der Relation bilden. Dey und Sarkar [5] verbessern dieses Modell, indem beliebige Schlüssel erlaubt werden. Es sind jedoch nur Projektionen erlaubt, welche auch den jeweiligen Schlüssel der angefragten Relation enthalten. In [4] wird für die Klasse der konjunktiven Anfragen ohne Selbstverbund sichere (d.h. semantisch korrekte) Ausführungspläne erzeugt. Die Ergebnisse von unsicheren Ausführungspläne werden approximativ angenähert. Keines dieser Systeme kann somit mit *beliebigen* Anfragen korrekt umgehen, da eine notwendige Normalisierung innerhalb des Auswertungsprozesses nicht durchgeführt wird.

$QSQL^P$ berechnet für beliebige Anfragen korrekte Wahrscheinlichkeiten. Bezüglich der einsetzbaren Wahrscheinlichkeitsmaße ist es jedoch z.B. gegenüber [2,8,21] restriktiver, da momentan keine *disjunkte* Eintrittswahrscheinlichkeiten auf Tupel- bzw. Attributebene unterstützt werden. Dieser Nachteil wird in [11] konzeptionell aufgehoben und soll in einer späteren Version von $QSQL^P$ umgesetzt werden.

Im Gegensatz zu extensionalen Ansätzen verarbeiten intensionale Systeme [6,8,21] während der Ergebnisberechnung Ereignisse oder Zufallsvariablen. Ab-

schließlich wird auf der Grundlage des finalen, normalisierten Ereignisses die eigentliche Ergebniswahrscheinlichkeit ermittelt. Dies garantiert wie in $QSQL^p$ die Berechnung von semantisch korrekten Ergebniswahrscheinlichkeiten. Für intentionale Systeme wurden verschiedene Approximationsverfahren entwickelt um die Wahrscheinlichkeitsberechnung auf Kosten der Ergebnissenauigkeit zu beschleunigen [15,17].

6.1 Logikbasierte Ähnlichkeitsbedingungen in probabilistischen Datenbanken

Neben der Art der Berechnung der Wahrscheinlichkeiten (extensional oder intensional) stellt sich vor allem die Frage der Ausdruckskraft bereits existierender Ansätze: Inwiefern ist es möglich in ihnen *beliebige logikbasierte Ähnlichkeitsanfragen* zu formulieren?

Insbesondere die wegweisenden Arbeiten [6] und [4] diskutieren explizit die Einbindung von Ähnlichkeitsprädikaten. Zur Abgrenzung gegenüber $QSQL^p$ soll deshalb auf diese beiden Ansätze im Detail eingegangen werden.

Ähnlichkeitsprädikate als Built-In-Prädikate Fuhr und Röllecke schlagen in [6] vor die Scoring-Funktion eines Ähnlichkeitsprädikates mit Hilfe einer eigenständigen probabilistischen Relation zu modellieren. Diese wird dann gemäß der ursprünglichen Anfragestruktur mittels einer Verbundoperation in den Anfrageausdruck integriert. Als Beispiel soll folgende Anfrage betrachtet werden: *Bestimme alle Vogelarten, welche in der Nähe von Berlin beobachtet worden sind.* Für das Ähnlichkeitsprädikat *Ort₁ in der Nähe von Ort₂* wird die probabilistische Relation SF_{OV} (Scoring-Funktion für Ortsvergleich) mit dem Relationenschema (Ort_1, Ort_2, Pr) und der Tupelmeng $SF_{OV} = \{(Cottbus, Berlin, 0.7), (Berlin, Berlin, 1.0)\}$ vereinbart. Die Tupel beinhalten die Auswertung der Ortsvergleiche zwischen Cottbus und Berlin, sowie Berlin und Berlin.

Der PRA-Algebraausdruck (siehe [6]) für die Beispielanfrage lautet: $VBeob \bowtie_{Ort=Ort_1} \sigma_{Ort_2=Berlin}(SF_{OV})$. Somit werden die Eintrittswahrscheinlichkeiten der Tupel aus $VBeob$ mit dem jeweiligen Ähnlichkeitswert des Ortsvergleichs aus SF_{OV} verbunden.

Problematisch bei diesem Vorgehen ist jedoch die Konstruktion von SF_{OV} . Sie verkörpert zwar ein Ähnlichkeitsprädikat, aber bezüglich der Auswertung stellt sie kein eigenständiges Konzept dar. Vielmehr unterliegt sie den gleichen Regeln, wie sie für alle probabilistische Relationen gelten. Somit müssen die Tupel unabhängige Basisereignisse darstellen damit die entsprechenden Aggregationsfunktionen angewendet werden können. Die Unabhängigkeit der Tupel ist in einer SF -Relation jedoch nicht gegeben. Fuhr und Röllecke schlagen deshalb vor, lediglich Anfragen zu benutzen, in denen keine Tupel aus gleichen SF -Relationen kombiniert werden. So darf z.B. eine bestimmte SF -Relation nicht mehr als einmal in einem Anfrageausdruck eingebunden werden und Projektionen können nicht mehr beliebig eingesetzt werden.

QSQL^P besitzt bezüglich der Anwendung von Ähnlichkeitsprädikaten mit unterschiedlichen Vergleichskonstanten eine vergleichbare Restriktion (siehe Kapitel 2.1), jedoch sind z.B. Projektionen innerhalb einer Anfrage *beliebig* anwendbar.

Ähnlichkeitsprädikate als Eintrittswahrscheinlichkeiten von Datenrelationen In [6] wurden Ähnlichkeitsprädikate als probabilistische Relationen modelliert, welche *während* des Auswertungsprozess eingebunden werden. Im Gegensatz dazu schlagen Dalvi und Suciu in [4] vor, die Wahrscheinlichkeiten für die verwendeten Ähnlichkeitsprädikate *vor* der eigentlichen Anfrageauswertung zu ermitteln. Die Ergebnisse dieser Vorberechnungen werden dann den Datenrelationen, auf welche sich die jeweilige Ähnlichkeitsprädikate beziehen direkt als Eintrittswahrscheinlichkeiten zu gewiesen. Zur Verdeutlichung sollen die bereits eingeführten Tabellen $VArt$ und $VBeob$ dienen, wobei die Tabelle $VBeob$ hier ohne die Spalte Pr betrachtet wird (notiert als $VBeob'$). Somit besitzen beide Relationen keine individuellen Eintrittswahrscheinlichkeiten.

Es soll folgende Beispielanfrage betrachtet werden: *Bestimme alle Vogelarten, welche in der Nähe von Berlin beobachtet worden sind und möglichst ähnlich einem Vorgabebild sind.* Als Algebraausdruck kann die Anfrage wie folgt formuliert werden: $\pi_{Art}(\sigma_{(Bild \approx_{BV} VBuild \wedge Ort \approx_{OV} Berlin)}(VArt \bowtie VBeob'))$. Bevor dieser Algebraausdruck ausgewertet wird, werden die Ähnlichkeitsprädikate $Bild \approx_{BV} VBuild$ bezüglich der Tupel in $VArt$ und das Ähnlichkeitsprädikat $Ort \approx_{OV} Berlin$ bezüglich der Tupel in $VBeob'$ berechnet. Die Ergebnisse werden als Eintrittswahrscheinlichkeiten in die Tabellen $VArt$ und $VBeob'$ kodiert. Die Tabellen $VArt$ und $VBeob'$ werden somit zu den probabilistischen Relationen $VArt^P$ und $VBeob^P$. Der auszuwertende Ausdruck ergibt sich dann zu $\pi_{Art}(VArt^P \bowtie VBeob^P)$.

Da in einer Verbundoperation die Wahrscheinlichkeiten für die zu verbindenden Tupel beider Relationen konjunktiv verknüpft werden [4], ergibt sich in der Ergebnisrelation die erwartete Wahrscheinlichkeit für die Konjunktion $Bild \approx_{BV} VBuild \wedge Ort \approx_{OV} Berlin$.

Dieser Mechanismus funktioniert jedoch lediglich bei Anfragen mit konjunktiv verknüpften Ähnlichkeitsprädikaten. Bereits bei einer einfachen Disjunktion von Ähnlichkeitsprädikaten, welche sich jeweils auf verschiedene Relationen beziehen, ist es nicht mehr möglich die Auswertung der disjunktiven Ähnlichkeitsbedingung aufzuteilen und in die jeweiligen Relationen zu verschieben. Beispielhaft soll folgende Anfrage betrachtet werden: *Bestimme alle Vogelarten, welche in der Nähe von Berlin beobachtet worden sind oder möglichst ähnlich einem Vorgabebild sind.* Der entsprechende Algebraausdruck ist nun gegeben durch: $\pi_{Art}(\sigma_{(Bild \approx_{BV} VBuild \vee Ort \approx_{OV} Berlin)}(VArt \bowtie VBeob'))$.

Ein Verschieben der Ähnlichkeitsprädikate in ihre jeweiligen Relationen steht der Widerspruch zwischen ihrer disjunkten Verknüpfung in der Selektion und der konjunktiven Verknüpfung von Wahrscheinlichkeiten innerhalb der Verbundoperation entgegen.

In weiteren Ansätzen (z.B. [21] und [8]) können Eintrittswahrscheinlichkeiten auch auf Attributebene modelliert werden. Somit besteht hier die Option die Auswertung der Ähnlichkeitsprädikate direkt in den abgefragten Attributen zu kodieren, bevor die eigentliche Abfrageauswertung gestartet wird. Dies funktioniert jedoch wiederum nur bei konjunktiv verknüpften Ähnlichkeitsprädikaten, da die Wahrscheinlichkeit für ein Tupel konjunktiv aus den einzelnen Eintrittswahrscheinlichkeiten seiner Attributwerte gebildet wird.

Zusammenfassend kann festgestellt werden, dass im Gegensatz zu $QSQL^P$ in den diskutierten Ansätzen [6], [4], [8] und [21] eine Integration *beliebiger logikbasierter Ähnlichkeitsbedingungen* nicht gegeben ist.

6.2 Fuzzy Datenbanken

Fuzzy Datenbanken (z.B. [7]) können ebenfalls mit unsicheren Anfragen auf unsicheren Daten umgehen. Es handelt sich hier jedoch nicht um ein probabilistisches Abfragemodell. Vielmehr werden die hier verwendeten Tupel-Zugehörigkeitswerte, ähnlich wie bei extensionalen probabilistischen Systemen, aggregiert ohne die eigentliche Semantik der kombinierten Teilbedingungen zu berücksichtigen. Das Konzept einer semantischen Normalisierung ist unbekannt.

Des Weiteren stellt die zu Grunde liegende Fuzzy Logik [22] im Allgemeinen keine Boolesche Algebra dar. Bekannte logische Äquivalenzen und Transformationsregeln (z.B. Idempotenz und Distributivität) sind somit nicht gültig. Ein detaillierter Vergleich zwischen Fuzzy Logik und Quantenlogik wird in [19] präsentiert.

7 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde die etablierte Many-World-Sematik für probabilistische Datenbanken um das Konzept der Relevanzwahrscheinlichkeiten erweitert. Diese werden in Form von logikbasierten Ähnlichkeitsanfragen auf einer unsicheren Datengrundlage formuliert. Neben der konzeptionellen Kombination beider Abfrageparadigmen wurde mit den Ähnlichkeitsanfragesprachen $CQQL^P$, QA^P und $QSQL^P$ eine praktische Umsetzung diskutiert. Des Weiteren wurde aufgezeigt, dass bisherige Ansätze beliebige logikbasierte Anfragen nicht ausreichend unterstützen. Als zukünftiges Forschungsvorhaben ist die Erweiterung des hier entwickelten probabilistischen Abfragemodells um disjunktive Eintrittswahrscheinlichkeiten auf Tupel- und Attributebene zu nennen.

Literatur

1. Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. On the Representation and Querying of Sets of Possible Worlds. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD Conference*, pages 34–48. ACM Press, 1987.
2. Daniel Barbará, Hector Garcia-Molina, and Daryl Porter. The management of probabilistic data. *IEEE Trans. Knowl. Data Eng.*, 4(5):487–502, 1992.

3. Roger Cavallo and Michael Pittarelli. The theory of probabilistic databases. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB*, pages 71–81. Morgan Kaufmann, 1987.
4. Nilesch Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal The International Journal on Very Large Data Bases*, 16(4):523–544, October 2007.
5. Debabrata Dey and Sumit Sarkar. A probabilistic relational model and algebra. *ACM Trans. Database Syst.*, 21(3):339–369, 1996.
6. Norbert Fuhr and Thomas Roelleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.
7. Jose Galindo, Angelica Urrutia, and Mario Piattini. *Fuzzy Databases: Modeling, Design and Implementation*. Idea Group Publishing, Hershey, USA, 2006.
8. Christoph Koch. MayBMS: A System for Managing Large Uncertain and Probabilistic Databases. In *Managing and Mining Uncertain Data*, chapter 6. Springer-Verlag, 2008.
9. Sebastian Lehrack. The Probabilistic Similarity Calculus CQQL^P. Technical report, Brandenburgische Technische Universität Cottbus, Institut für Informatik, Cottbus, Germany, 2010.
10. Sebastian Lehrack. The Retrieval Model Behind CQQL. Technical report, Brandenburgische Technische Universität Cottbus, Institut für Informatik, 2010.
11. Sebastian Lehrack. A Unifying Probability Measure for Logic-Based Similarity Conditions on Uncertain Relational Data. Technical report, Brandenburgische Technische Universität Cottbus, Institut für Informatik, Cottbus, Germany, 2011.
12. Sebastian Lehrack and Sascha Saretz. The Definition of QA^P. Technical report, Brandenburgische Technische Universität Cottbus, Institut für Informatik, Cottbus, Germany, 2010.
13. Sebastian Lehrack and Ingo Schmitt. QSQL: Incorporating Logic-Based Retrieval Conditions into SQL. In Hiroyuki Kitagawa, Yoshiharu Ishikawa, Qing Li, and Chiemi Watanabe, editors, *DASFAA (1)*, volume 5981 of *Lecture Notes in Computer Science*, pages 429–443. Springer, 2010.
14. Sebastian Lehrack, Ingo Schmitt, and Sascha Saretz. CQQL: A Quantum Logic-Based Extension of the Relation Domain Calculus. In *Proceedings of the International Workshop Logic in Databases (LID '09)*, October 2009.
15. Dan Olteanu, Jiewen Huang, and Christoph Koch. Approximate confidence computation in probabilistic databases. In *ICDE*, pages 145–156, 2010.
16. J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.
17. Christopher Re, Nilesch N. Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895, 2007.
18. Ingo Schmitt. QQL: A DB&IR Query Language. *The VLDB Journal*, 17(1):39–56, 2008.
19. Ingo Schmitt, Andreas Nürnberger, and Sebastian Lehrack. On the Relation between Fuzzy and Quantum Logic. In *Views on Fuzzy Sets and Systems from Different Perspectives*, chapter 5. Springer-Verlag, 2009.
20. C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.
21. J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR '05)*, January 2005.
22. Lotfi Asker Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, June 1965.

Generierung maßgeschneiderter Relationenschemata in Softwareproduktlinien mittels Superimposition

Martin Schäler¹, Thomas Leich², Norbert Siegmund¹, Christian Kästner³, und Gunter Saake¹

¹ Fakultät für Informatik, Universität Magdeburg, Deutschland
{schaeler, nsiegmun, saake}@cs.uni-magdeburg.de,

² METOP Forschungsinstitut, Magdeburg, Deutschland
thomas.leich@metop.de

³ Fachbereich Mathematik und Informatik, Philipps Universität Marburg, Deutschland
kaestner@informatik.uni-marburg.de

Zusammenfassung Die Erstellung eines individuellen Programms aus einer Softwareproduktlinie (Programmfamilie) erfordert auf Anwendungs- und Datenbankseite einen speziell angepassten und aufeinander abgestimmten Funktionsumfang. Die Modellierung maßgeschneiderter Relationenschemata stellt z.B. aufgrund der großen Anzahl an Programmen, die aus einer Produktlinie erstellt werden können, eine Herausforderung dar. Wir präsentieren einen Lösungsvorschlag zur Modellierung und Generierung von maßgeschneiderten Relationenschemata mittels Superimposition. Wir zeigen anhand einer realen, produktiv eingesetzten Fallstudie welche Vorteile unser Ansatz in den Bereichen Wartung und Weiterentwicklung erzeugt und welche Herausforderungen beispielsweise durch redundant definierte Schemaelemente existieren.

1 Einleitung

Die Entwicklung von langlebigen Datenbank (DB)-Schemata stellt seit jeher eine Herausforderung dar [21]. Moderne Anforderungen an Softwareimplementierungen in Bezug auf Wiederverwendbarkeit und individualisierten Funktionsumfang eröffneten auf Anwendungsseite ein weites Forschungsgebiet und führten zu einer Vielzahl von Lösungsvorschlägen [11]. Hingegen sind die Auswirkungen im DB-Schemabereich weitestgehend unbekannt [26]. Vor allem im Kontext von Softwareproduktlinien [19] (SPL) entstehen neue Herausforderungen für die Entwicklung von DB-Schemata. SPLs werden häufig zur Erstellung individuell auf den Nutzer zugeschnittener Produkte eingesetzt [2]. In ihr repräsentieren *Merkmale* eine für den Kunden sichtbare Funktion der Software [11]. Eine SPL zur Erstellung eines maßgeschneiderten Dokumentenverwaltungssystems kann beispielsweise die Merkmale *Nutzerverwaltung* und *Mehrbenutzersynchronisation* enthalten, die unabhängig als Extrafunktionalität vermarktet werden. Entsprechend der Kundenwünsche wird die Dokumentenverwaltungssoftware mit oder ohne diese Merkmale generiert (Variante der Software). Je nach Funktion des Merkmals ist es auf die Existenz gewisser DB-Schemaelemente zur Speicherung und Verarbeitung der für das Merkmal relevanten Daten angewiesen. Die Nutzerverwaltung des Dokumentenverwaltungssystems benötigt z.B. Relationen in denen die Nutzer mit den zugehörigen Rechten

hinterlegt sind. Häufig wird bisher ein globales Schema unabhängig von der Merkmalauswahl ausgeliefert. Die derzeit verwendeten Vorgehen erzeugen die folgenden Probleme:

- Es wird ein *komplexes, schwer verständliches Gesamtschema* erzeugt, von dem große Teile in vielen Varianten nicht benötigt werden. Dies erschwert die *Wartbarkeit* eines Variantschemas und die *Weiterentwicklung* der Software.
- Die *Ausdrucksfähigkeit der Modellierung* wird beschränkt. Es können beispielsweise Relationen in eine variable Anzahl von Fragmenten partitioniert werden oder es kann kein Gesamtschema angegeben werden, wenn die Modellierung konkurrierende Schemaelemente enthält.
- Es entstehen Probleme in Bezug auf die *Integrität* der enthaltenen Daten.
- Die *Skalierbarkeit* des Ansatzes muss gewährleistet sein, da die Anzahl der generierbaren Varianten exponentiell zur Zahl der Merkmale wächst.

Wir sind hingegen daran interessiert, für jede Variante der Software ein maßgeschneidertes DB-Schema zu generieren. Langfristig beabsichtigen wir die folgenden Zielstellungen zu erreichen:

- Ziel ist es den *Funktionsumfang* des DB-Schemas *individuell* auf den Kunden abstimmen. Dazu gehört u.a. die *Optimierung* auf bestimmte Anfragetypen (Exact Match, Range Queries, etc.) oder die Einbeziehung von *Sicherheitsaspekten*, wie Verschlüsselung der Daten oder Information Hiding.
- Ähnlich zur Anwendungsseite sollen DB-Elemente in Varianten der SPL *wiederverwendet* werden. Das bedeutet, dass Teile der Modellierung auf DB-Schemaebene in ähnlichen Produkten erneut verwendet werden, um den *Aufwand* zur Erstellung von Varianten zu verringern.
- Die *Verständlichkeit* der Modellierung sowie einzelner Variantschemata der Software soll verbessert werden, um sowohl die *Weiterentwicklung* der Software, als auch die *Wartung* von Varianten zu vereinfachen.

In einem ersten Schritt zur Erfüllung der Ziele wird untersucht, inwiefern sich Varianten eines Schemas für eine reale Fallstudie in einem konkreten Datenmodell erstellen lassen. Weiterhin wird überprüft welche Probleme dabei auftreten, um daraus Erkenntnisse abzuleiten, inwiefern die Komplexität eines solchen Vorgehens einen Einsatz in der Praxis erlaubt. Aufbauend auf den Ergebnissen von SIEGMUND et. al. [26], die auf der BTW 2009 einen ersten Ansatz zur Erstellung maßgeschneiderter ER-Schemata [8] präsentierten, schlagen wir ein solches Vorgehen für das Relationenmodell [10] vor. Hierbei handelt es sich unseres Wissens nach um die erste Arbeit ein maßgeschneidertes DB-Schema für ein konkretes Datenmodell und für eine reale, produktiv eingesetzte Fallstudie zu erzeugen, um die Praxistauglichkeit des Vorgehens besser beurteilen zu können. Wir untersuchen die folgenden Problemstellungen:

1. Wie können variable DB-Schemata für das Relationenmodell im Kontext von SPLs modelliert werden?
2. Wie lässt sich aus der Modellierung ein maßgeschneidertes DB-Schema für eine Variante der SPL erstellen?

3. Inwiefern existieren Interaktionen von Merkmalen auf DB-Schemaebene und welche Auswirkungen haben sie auf die Erzeugung eines Variantenschemas?
4. In welchem Umfang zeigen sich die erkannten Probleme bei der Anwendung unseres Lösungsvorschlags an einer produktiv eingesetzten Fallstudie?

2 Die Fallstudie ViT-Manager

Bei der Fallstudie ViT®-Manager⁴ handelt es sich um ein Controlling-Tool mit dessen Hilfe es möglich ist Verbesserungspotentiale in einem Unternehmen zu erfassen, zu klassifizieren und die Ausschöpfung des Potentials nachzuvollziehen. Die Fallstudie verfügt ohne externe Bibliotheken über ca. 50.000 Zeilen Quellcode, das DB-Schema enthält 54 Relationen mit 309 Attributen. Sie wird produktiv sowohl in Mittelstandsunternehmen, als auch in Großkonzernen eingesetzt. Weiterhin sind die einsetzenden Unternehmen in verschiedenen Branchen angesiedelt. Aus den heterogenen Einsatzgebieten ergeben sich unterschiedlichste Anforderungen an die einzelnen Varianten der Fallstudie. In Großkonzernen stehen beispielsweise das Aufdecken von Synergien, umfangreiche Reportingfunktionen sowie die Unterstützung möglichst vieler *Methoden zur Ausschöpfung der vorhandenen Verbesserungspotentiale (MAP)* im Vordergrund. Hingegen legen Mittelstandsunternehmen vor allem Wert auf eine konsequente und nachvollziehbare Ausschöpfung der aufgedeckten Potentiale anhand monetärer Größen, wie z.B. eingesparter Arbeitszeit.

2.1 Die ViT-SPL

Die Fallstudie ViT-Manager entstand aus einer monolithischen Anwendung, die aufgrund von neuen Kundenanforderungen durch zusätzliche Funktionen erweitert wurde. Die zunehmende Komplexität und die unterschiedlichen Anforderungen der Kunden ließen daher eine Überführung der monolithischen Anwendung in eine SPL sinnvoll erscheinen. Bei einer SPL handelt es sich um eine Familie von Programmen, die eine Anzahl gleicher und unterschiedlicher Merkmale besitzt [9] und häufig zur Erstellung individueller Software genutzt wird [2]. Ein Merkmal beschreibt dabei eine für den Nutzer wesentliche Funktion der Software [11]. Die Reportingfunktion oder die Nutzerverwaltung des ViT-Managers sind Beispiele für ein solches Merkmal. Die Generierung eines individuellen Produktes erfolgt durch Auswahl der benötigten Merkmale. Das Produkt wird anschließend durch eine zuvor definierte Erstellungsstrategie generiert. In ViT erfolgt dies durch eine Plugin-Architektur [13].

2.2 Das Merkmalmodell der Fallstudie

Die zur Verfügung stehenden Merkmale und deren Beziehungen untereinander werden im Merkmalmodell zusammengefasst und im Merkmaldiagramm visualisiert [14]. Das Modell beschreibt die Variabilität einer SPL. Es können z.B. Beziehungen zwischen Merkmalen und Bedingungen, wie Merkmal *A benötigt B*, definiert werden. Die Variabilität einer SPL besagt, welche Kombinationen von Merkmalen zu einer gültigen

⁴ Verbesserung im Team (ViT) ist ein eingetragenes Warenzeichen der Karer Consulting AG.

Variante führen und somit welche Funktionen miteinander kombiniert werden können. Die Anwendung wurde, wie in Abb. 1 gezeigt, in Merkmale zerlegt um Varianten der

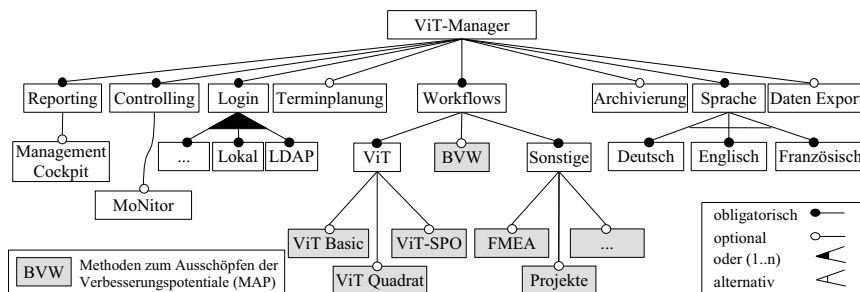


Abbildung 1. Ausschnitt des Merkmaldiagramm des ViT-Managers Version 1.6.9b

SPL generieren zu können. Kern der Anwendung ist die Unterstützung der verschiedenen *MAPs*, die in einem eigenem Merkmal implementiert und als Blätter des Workflowteilbaums zu erkennen sind. Es handelt sich um optionale Merkmale, die nicht in jeder Variante der Software enthalten sein müssen. Weiterhin kann der ViT-Manager entweder auf *Deutsch*, *Englisch* oder *Französisch* ausgeliefert werden, unterstützt auf Wunsch ein selbst definierbares *Reporting* und verfügt über weitere optionale Funktionen wie *Archivierungsfunktionen*, bei denen die Daten zusätzlich anonymisiert werden können. Die Anmeldung eines Nutzers an die Software ist auf verschiedenen Wegen, wie *LDAP* oder *Lokaler Anmeldung*, möglich, von denen mehrere gleichzeitig in einer Variante enthalten sein können. Insgesamt können mehrere tausend valide Varianten der SPL erstellt werden.

Bei der Zerlegung der Anwendung in Merkmale wurde festgestellt, dass die Verwendung eines monolithischen DB-Schemas Probleme wie mangelnde Übersichtlichkeit, ungenutzte Schemaelemente und die fehlende Unterstützung alternativer Merkmale mit sich bringt. Die Probleme auf DB-Schemaebene werden nachfolgend allgemeingültig beschrieben, bevor wir in Abschnitt 6 zur Fallstudie zurückkehren, wobei sie solange als durchgängiges Beispiel genutzt wird.

3 Problembeschreibung

Unterschiedliche Anwender haben verschiedene Anforderungen an Softwareprodukte. Diese werden auf Anwendungsseite beispielsweise durch Varianten einer SPL erfüllt. Es existiert eine Vielzahl von Ansätzen, die die Generierung von Varianten einer Software z.B. mittels plugin-basierten Frameworks [13] erlaubt. Hingegen sind die bisherigen Untersuchungen auf DB-Schemaebene nicht ausreichend [26]. Bei DB-Schemata handelt es sich um langlebige Softwareprodukte, die nicht selten länger genutzt werden als die Anwendungen in deren Kontext sie entwickelt wurden [21]. Somit kommt ihrer Entwicklung besondere Bedeutung zu.

3.1 Grenzen derzeitiger Ansätze

Nachfolgend werden die derzeit verwendeten Lösungsansätze vorgestellt und erläutert welche in der Einleitung genannten Probleme bei ihnen auftreten.

Verwendung eines globalen Schemas. Häufig wird bisher für jede Variante einer SPL ein statisches, globales und historisch gewachsenes DB-Schema unabhängig von der Merkmalauswahl ausgeliefert [26]. Es enthält sämtliche Schemaelemente, die jemals in einer der Varianten benötigt werden. Bei entsprechender Komplexität der Software entsteht somit ein *hochkomplexes, schwer verständliches* DB-Schema. Die Auslieferung eines monolithischen Schemas kann dazu führen, dass in einer Variante Schemaelemente enthalten sind, die nicht benötigt werden. Somit entstehen, je nach Merkmalauswahl, ungenutzte Tabellen und Attribute. Dies führt zu *Integritätsproblemen* bei leeren Spalten auf denen beispielsweise Fremdschlüssel, Check oder Not Null Bedingungen definiert sind oder es wird gänzliche auf Schemaelemente zur Integritätssicherung verzichtet. Darüber hinaus erschwert ein solches globales Schema das Verständnis und beeinträchtigt die *Wartung und Weiterentwicklung*. Ein weiterer entscheidender Nachteil besteht darin, dass nicht in jedem Fall ein globales Schema angegeben werden kann. Dies ist der Fall, wenn in einer SPL alternative, sich gegenseitig ausschließende Merkmale existieren. Diese können sich *widersprechende Schemaelemente* definieren, die nicht gleichzeitig in einem globalen Schema enthalten sein können.

Sichtbasierte Ansätze. Das Problem der fehlenden Variabilität auf DB-Schemaebene wurde bereits erkannt und Sichten aus dem globalen Schema generiert, die den Varianten einer SPL entsprechen [7]. Das globale Schema wird dennoch in jeder Variante der SPL ausgeliefert. Die Generierung solch maßgeschneiderter Sichten verringert die Komplexität eines Schemas nicht, sondern verbirgt sie lediglich. Die Sichten *erhöhen die Komplexität* des Schemas für den Entwickler und während der Wartung, da sie zusätzliche Schemaelemente einführen, anstelle nicht benötigte zu entfernen. Somit wird die Verständlichkeit des Gesamtschemas negativ beeinflusst.

Lösung in Frameworks. In Frameworks werden häufig Namensräume verwendet, die verhindern, dass die einzelnen Plugins, welche als Merkmale angesehen werden können, sich gegenseitig beeinflussen. Somit können plugin-übergreifend *keine Integritätsbedingungen* verwendet werden oder es werden zusätzliche Abhängigkeiten wie Plugin A benötigt B erzeugt. Die Sicherung der Konsistenz erfolgt daher meist auf Anwendungsebene [26]. Aufgrund von Fehlern in der Anwendung oder durch direkten Zugriff auf die DB ist es daher dennoch möglich inkonsistente Daten in der DB zu hinterlegen. Weiterhin führt dieses Vorgehen dazu, dass jedes Plugin für eine semantisch identische Relation eine *einzelne Partition* dieser anlegt. Legen beispielsweise drei optionale Merkmale horizontale Partitionen der Relation Mitarbeiter z.B. für verschiedene Unternehmensbereiche an, ist die Ausgabe der Gesamtmitarbeiterzahl des Unternehmens nicht trivial. Ein Reportingmerkmal müsste je nachdem welche Merkmale in der Variante enthalten sind, auf die einzelnen Partitionen zugreifen. Es muss dazu wissen, in welchen Merkmalen solche Partitionen angelegt werden und welche davon in der Variante existieren. Ein solches Vorgehen erzeugt daher einen deutlich erhöhten Aufwand auf Anwendungsseite.

3.2 Modellierung eines maßgeschneiderten Schemas

Ein intuitiver Ansatz zur Modellierung maßgeschneiderter DB-Schemata besteht darin für jede Variante der Software ein eigenes Relationenschema zu modellieren. Der Aufwand eines solchen Vorgehens ist in der Praxis jedoch nicht zu beherrschen. Die Zahl der Varianten wächst exponentiell zur Anzahl der Merkmale. Die Fallstudie ViT verfügt über 26 Merkmale aus denen sich mehrere tausend valide Varianten generieren lassen. Daher ist es nicht möglich für jede Variante der Software ein eigenes Schema zu entwerfen, sondern es müssen Teile der Modellierung wiederverwendet werden können.

Abgrenzung der Betrachtung. In unserem Ansatz wird ein maßgeschneidertes Relationenschema erstellt, hierfür wird ein Ausschnitt des Relationenmodells betrachtet. In der Modellierung sind vorerst die folgenden Elemente der Datendefinitionssprache des SQL:1999 Standards [1] enthalten: *Relationen, Attribute, der Wertebereich eines Attributs und die Definition des Primärschlüssels einer Relation*. Weiterhin wird zur Vereinfachung der Betrachtung davon ausgegangen, dass die Modellierung der ersten Normalform genügt. Es ist geplant zukünftig weitere Schemaelemente zu unterstützen. Wir konzentrieren uns in dieser Arbeit auf das DB-Schema, weitere Themen wie beispielsweise Variabilität auf Datenebene werden nicht betrachtet.

3.3 Generierung eines Variantenschemas

Aus der Modellierung soll das individuell zugeschnittene DB-Schema erzeugt werden. Es muss eine Technik zur Generierung des Variantenschemas gewählt werden, welche die folgenden Anforderungen erfüllt.

Vollständigkeit. Alle Schemaelemente, die von den gewählten Merkmalen benötigt werden, müssen im Schema der Variante enthalten sein.

Komplexitätsreduktion. Es sollen ausschließlich benötigte Schemaelemente im Variantenschema vorhanden sein, um die Wartung dieses zu vereinfachen.

Komponierbarkeit. Ein Merkmal muss die Modellierung eines anderen erweitern können. Insbesondere soll es möglich sein Attribute zu Relationen hinzuzufügen ohne dass bekannt ist, in welchen Merkmalen die Relation definiert wurde, so dass keine Kenntnisse über den Aufbau der anderen Merkmale vorhanden sein müssen. Dies erleichtert sowohl die Weiterentwicklung als auch die Wartung.

Korrektheit. Alle Schemaelemente müssen in der korrekten Ausprägung im Variantenschema vorhanden sein. Das bedeutet, dass alle Attribute eines Merkmals den in der Modellierung definierten Wertebereich erhalten und sie Bestandteil des Primärschlüssels der Relation sind, wenn dies in der Modellierung angegeben wurde.

3.4 Interaktion von Merkmalen auf DB-Schemaebene

Das Problem der Interaktion von Merkmalen ist ein wohlbekanntes Problem auf Anwendungsseite und schränkt die Wiederverwendbarkeit von Softwareartefakten ein [15,20]. Es entsteht, wenn zwischen mehreren im Merkmalmodell unabhängigen Merkmalen, aufgrund der Implementierung, Abhängigkeiten bestehen. Das bedeutet, dass einige laut

Merkmalmodell valide Varianten der SPL durch die zusätzlichen Abhängigkeiten der Implementierung nur dann generiert werden können, wenn die zusätzlichen Abhängigkeiten aufgelöst werden. Bezogen auf das Beispiel des Dokumentenverwaltungssystems benötigt das Merkmal *Nutzerverwaltung*, nur dann wenn ebenfalls das optionale Merkmal *Mehrbenutzersynchronisation* gewählt ist, zusätzliche Funktionen. Es muss beispielsweise feststellen können, welche Nutzer (exklusive) Schreibsperrern auf Dokumenten aufheben können. Es wird daher auf folgende Fragestellungen eingegangen:

- Existiert das Problem der Interaktion von Merkmalen auf Modellierungs- oder Implementierungsebene des DB-Schemas in unserem Lösungsansatz?
- Welchen Einfluss hat es auf die Generierung von Variantenschemata?
- Wie und in welchem Umfang äußert sich dieses Problem in der Fallstudie?

4 Modellierung eines maßgeschneiderten DB-Schemas

In unserem Ansatz wird für jedes Merkmal festgehalten, welche Schemaelemente ein Merkmal benötigt und aus dieser Modellierung für jede Variante ein maßgeschneidertes Schema komponiert. Die Idee besteht darin, dass jedes Merkmal dem Variantenschema die zusätzlich von ihm benötigten Schemaelemente hinzufügt und mehrfach definierte Elemente dennoch lediglich einmal im Ergebnis vorhanden sind. Somit muss für jedes Merkmal ein Teilschema modelliert werden und nicht für jede Variante ein komplettes DB-Schema.

4.1 Zerlegung des DB-Schemas in Merkmale

Es existieren zwei Möglichkeiten ein variables Schema zu modellieren. Es kann von Anfang an als variables Schema entworfen werden oder es wird aus einem zuvor verwendeten globalen Schema erzeugt. In der Fallstudie ist bereits ein monolithisches Gesamtschema vorhanden, deshalb konzentrieren wir uns auf diesen Fall. Bei der Restrukturierung der Anwendung in eine SPL muss daher ebenfalls das DB-Schema in Merkmale untergliedert werden. Zu diesem Zweck wird nachfolgend ein allgemeingültiger Zerlegungsprozess vorgestellt. Der von uns vorgeschlagene Prozess untergliedert sich in die folgenden Teilprobleme.

1. Bestimmung der Merkmale in die das Schema zerlegt wird.
2. Definition eines Kriteriums, wann ein Schemaelement einem Merkmal zugeordnet wird.
3. Ausführen der Zerlegung anhand des zuvor bestimmten Kriteriums.

Bestimmung der Merkmale in die das Schema zerlegt wird. Auf DB-Schemaebene werden dieselben Merkmale, wie auf Anwendungsseite verwendet, da für die Anwendung ein maßgeschneidertes Schema erstellt werden soll. Somit enthält ein Merkmal sowohl die Implementierung des Anwendungs-, als auch die des DB-Teils einer für den Nutzer wichtigen Funktionalität. Eine Variante der Software kann wie bisher durch Auswahl der notwendigen Merkmale im Merkmalmodell der SPL erfolgen und muss nicht für Anwendungs- und DB-Teil separat erfolgen [24].

Wann wird ein Schemaelement einem Merkmal zugeordnet? Aus der Anforderung, dass für jede Variante einer Software ein maßgeschneidertes DB-Schema erstellt werden soll, ergibt sich, dass in dem Teilschema eines Merkmals alle Schemaelemente vorhanden sein müssen, die das Merkmal zur korrekten Funktionsweise benötigt.

Definition 1. *Ein Schemaelement wird einem Merkmal genau dann zugeordnet, wenn das Element innerhalb des merkmalspezifischen Quellcodes lesend oder schreibend verwendet wird.*

Das Merkmal ViT Basic trägt beispielsweise die Beschreibung eines Verbesserungspotentials in das Attribut *Beschreibung* der Relation *probleme* ein, daher wird die Relation mit dem genannten Attribut in die Modellierung des Merkmals aufgenommen. Vorkommen von *SELECT* *-Anweisungen werden durch die Signatur sämtlicher in der FROM-Klausel aufgeführten Relationen ersetzt. Analog wird bei *INSERT INTO* Statements ohne Attributliste verfahren.

Ausführen der Zerlegung. Nach Definition 1 müssen alle DB-Aufrufe der Software identifiziert und den entsprechenden Merkmalen zugeordnet werden. Dies kann entweder über spezielle Eigenschaften der Implementierung auf Anwendungsseite der SPL, die das Auffinden solcher Stellen im Quellcode erleichtern oder durch statische Analyse der Anwendung [25] erfolgen. In der Fallstudie erfolgt jeder DB-Zugriff über das global definierte DB-Objekt, so dass die Zugriffe mittels der vorhandenen Suchfunktion der verwendeten Entwicklungsumgebung identifiziert werden können.

4.2 Modellierung eines Merkmals

Die Modellierung des Schemas eines Merkmals erfolgt mit Hilfe einer einfachen textuellen Syntax. In der Definition 1 wird erläutert, wann ein Schemaelement in die Modellierung eines Merkmals aufgenommen wird. Das Kriterium stellt sicher, dass alle benötigten Schemaelemente in ihr vorhanden sind.

In der Abb. 2 ist die Syntaxrepräsentation der Merkmale *Login* und *Lokale Anmeldung* aus Abb. 1 angegeben. Die Modellierung des Merkmals *Login* enthält eine Relation *login_daten* mit den Attributen *deaktiviert*, *loginname* und *personalnummer* mit den jeweiligen Wertebereichen. Über den Primärschlüssel *Personalnummer* wird der *Loginname* eindeutig einem Benutzer der Software zugeordnet. Der eigentliche Passwortabgleich erfolgt jedoch durch das Merkmal *Lokale Anmeldung* indem das für den Loginnamen angegebene Passwort mit dem in der Datenbank hinterlegten verglichen wird. Daher benötigt das Merkmal ebenfalls eine Relation *login_daten* und legt die beiden Attribute *loginname* und *passwort* an. Somit ist z.B. das Attribut *loginname* redundant in den Teilschemata beider Merkmale vorhanden.

Alternativ dazu könnte für jedes Merkmal eine eigene Relation angelegt werden. Dies führt jedoch zu den selben Problemen, wie sie in 3.1 für Frameworks beschrieben wurden.

4.3 Interaktionen auf Modellierungsebene

Interaktionen auf Modellierungsebene existieren in zwei Formen, die nachfolgend näher erläutert werden. Sie treten auf, wenn

Login		Lokale Anmeldung	
1	RELATION login_daten(1	RELATION login_daten(
2	deaktiviert int(1) ,	2	loginname varchar(255) ,
3	loginname varchar(255) ,	3	passwort varchar(255)
4	personalnummer varchar(255) PK	4);
5);		

Abbildung 2. Modellierung zweier Merkmale mit redundanten Schemaelementen

- semantisch identische Schemaelemente von mehreren Merkmalen redundant definiert werden;
- aufgrund der Kombination der Merkmale die Modellierung eines Merkmals angepasst werden muss, da zusätzliche Schemaelemente benötigt werden.

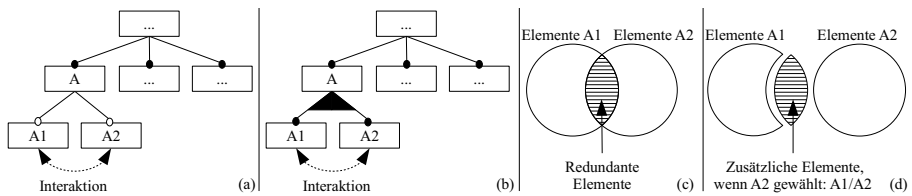


Abbildung 3. Interaktionen auf Modellierungsebene

Redundant definierte Schemaelemente. Redundant definierte Schemaelemente treten auf, wenn beispielsweise zwei zueinander optionale Merkmale (Abb. 3 (a)) oder Merkmale, die über eine oder-Beziehung (Abb. 3 (b)) miteinander verbunden sind, dieselben Schemaelemente benötigen. Da Varianten existieren in denen lediglich eines der beiden Merkmale vorhanden ist, muss jedes Merkmal alle benötigten Schemaelemente in seiner Modellierung enthalten. Redundanzen treten in dem hier betrachteten Ausschnitt des Relationenmodells auf Attributebene auf, wenn zu einer Relation dasselbe Attribut in mehreren Merkmalen definiert wird. Mehrfach definierte Relationen können unterschiedliche Attribute enthalten, so dass dieser Fall nicht als redundante Definition angesehen wird. In ViT existieren solche Interaktionen beispielsweise zwischen den Blättern des Workflowteilbaums aus Abb. 1. Hierbei handelt es sich um die Merkmale, welche die Implementierung der verschiedenen *MAPs* enthalten. Sie sind zueinander optional und enthalten eine große Anzahl an redundanten Schemaelementen (Abb. 3 (c)). Redundante Schemaelemente erzeugen die folgenden Probleme.

- Für jede redundante Definition muss sichergestellt werden, dass jedes Attribut denselben Wertebereich erhält und keine widersprüchlichen Modellierungen entstehen, aus denen kein valides Schema generiert werden kann.
- Die redundanten Schemaelemente erhöhen die Komplexität des Schemas eines Merkmals und beeinträchtigen somit die Verständlichkeit der gesamten Modellierung.

Zusätzlich benötigte Schemaelemente. Bei einer bestimmten Kombination von gewählten Merkmalen ist es, aufgrund der Interaktion dieser Merkmale, notwendig zusätzliche Schemaelemente zum Variantenschema hinzuzufügen. In der Fallstudie existiert beispielsweise eine Funktion zur *Archivierung* der Daten der einzelnen *MAPs*. Je nachdem welche *MAPs* in der Variante vorhanden sind, müssen die entsprechenden Archivtabellen angelegt werden. Die Lösung alle theoretisch benötigten Schemaelemente in die Modellierung der *Archivierung* aufzunehmen erzeugt die in 3.1 beschriebenen Probleme eines globalen Schemas. In unserem Ansatz werden die zusätzlich benötigten Schemaelemente (siehe Abb. 3 (d)) der Modellierung des Merkmals durch *Derivatives* [17] hinzugefügt. Hierbei handelt es sich um zusätzliche Merkmale, die dem Variantenschema automatisch hinzugefügt werden, wenn die Merkmale auf die sie sich beziehen gleichzeitig in der Variante enthalten sind. Das Derivative *A1/A2* wird dem Variantenschema automatisch hinzugefügt, wenn die Merkmale *A1* und *A2* in einer Variante vorhanden sind.

5 Generierung eines Variantenschemas

Es wurde bisher festgehalten, dass ein Merkmal auf DB-Schemaebene ein relationales Teilschema enthält. Die Generierung einer Variante der Software erfolgt durch Auswahl der benötigten Merkmale anhand der zuvor festgelegten Erstellungsstrategie. Nachfolgend wird veranschaulicht, wie die Komposition eines Variantenschemas mittels Superimposition (Überlagerung) in unserem Lösungsvorschlag erfolgt. Es wird weiterhin untersucht welche Probleme während der Komposition auftreten können.

5.1 Superimposition - Sprachunabhängiger Kompositionsalgorithmus

Die Superimposition beschreibt einen sprachunabhängigen Kompositionsalgorithmus, bei dem mehrere Merkmale durch Mischen ihrer zugrunde liegenden (Sub) Strukturen vereinigt werden [3]. Diese Technik wird beispielsweise in der Sichtintegration [5] und der merkmal-orientierten Softwareentwicklung [6] verwendet. Für die Superimposition von Merkmalen (\bullet -Operator) existiert von APEL et al. eine Algebraformalisierung die zeigt, dass die Operation assoziativ und im allgemeinen Fall nicht kommutativ ist [4]. Ein Variantenschema (S) wird mittels Superimposition aller gewählten Merkmale (m_i) durch den Operator \bullet erzeugt und repräsentiert selbst wieder ein Merkmal (M):

$$\bullet: M \times M \rightarrow M \qquad S = m_n \bullet \dots \bullet m_2 \bullet m_1 \qquad (1)$$

Struktur eines Merkmals auf DB-Schemaebene. Die Komposition zweier Merkmale basiert auf der Vereinigung von *Merkmalstrukturbäumen* (MSB), welche die interne Struktur eines Merkmals repräsentieren und als vereinfachter abstrakter Syntaxbaum angesehen werden können [3]. In der Abbildung 4 ist der MSB des Merkmals *Login* basierend auf der Modellierung aus Abb. 2 abgebildet. Die Wurzel des Baumes verdeutlicht, dass es sich um die Modellierung des DB-Schemas handelt, somit kann das Merkmal sowohl das DB-Schema, als auch die Anwendungsimplementierung enthalten. Die Kinder des Wurzelknotens enthalten alle vom Merkmal benötigten Relationen. Im

angegebenen Beispiel definiert das Merkmal die Relation *login_daten*. Auf der Blattebene befinden sich die Attribute der Relationen. Sie verweisen auf die Implementierung des Schemaelementes. Sie enthält den Wertebereich und die Angabe ob das Attribut Teil des Primärschlüssels der Relation ist. Der Grafik ist zu entnehmen, dass das Merkmal Login die Relation *login_daten* sowie die Attribute *deaktiviert*, *loginname* und *personalnummer* enthält. Die Implementierung des Attributs *personalnummer* definiert den Wertebereich als *varchar(255)* und zeigt, dass das Attribut einziger Bestandteil des Primärschlüssels der Relation ist.

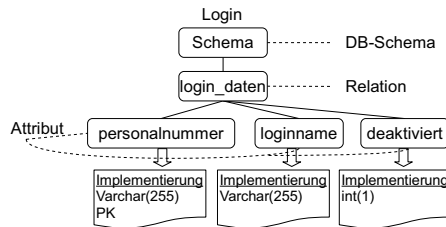


Abbildung 4. Merkmalstrukturbaum des Login-Merkmals

5.2 Vereinigung zweier Merkmale

Die Superimposition zweier Merkmale erfolgt, wenn beide Merkmale ausgewählt wurden und daher Bestandteile der zu erstellenden Variante sind. Der Kompositionsprozess beginnt an den Wurzeln der zu vereinigenden MSBs und erfolgt dann rekursiv im gesamten Baum. Zwei Knoten werden miteinander verschmolzen, wenn ihre Elternknoten miteinander verschmolzen wurden bzw. es sich um Wurzelknoten handelt und sowohl Name als auch Typ übereinstimmen. In der Abb. 5 wird das Verschmelzen zweier Merkmale in Gleichung (2) am Beispiel der bereits bekannten Merkmale *Login* und *Lokale Anmeldung* veranschaulicht, welche als Teil einer zu erstellenden Variante ausgewählt wurden.

$$m_{\text{Lokale Anmeldung}} \bullet m_{\text{Login}} = m_{\text{Ergebnis}} \quad (2)$$

Das Ergebnis besteht aus dem Teilschema des Merkmals *Login*, welches um das zusätzliche Attribut *passwort* des Merkmals *Lokale Anmeldung* ergänzt wurde. Das redundant definierte Attribut *loginname* ist im Ergebnis, wie beabsichtigt, einmal enthalten. Problematisch ist der Fall, wenn auf Blattebene ein Attribut mehrfach definiert ist. Dieser Fall ist im Beispiel 5 durch das Attribut *loginname* gegeben, es repräsentiert das Problem der Interaktion von Merkmalen auf Implementierungsebene (siehe 3.4).

5.3 Interaktionen auf Implementierungsebene: Verschmelzung von Blattknoten

Das Problem der Interaktion von Merkmalen tritt auf Implementierungsebene in unserem Ansatz auf, wenn Attribute redundant definiert wurden (siehe 4.3). Für das Problem der

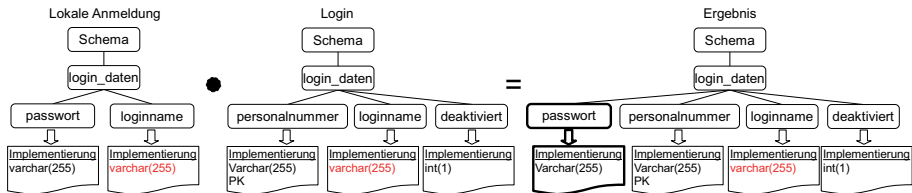


Abbildung 5. Superimposition: Beispiel für Merkmale mit Redundanzen

Verschmelzung von Blattknoten besteht die Möglichkeit entweder die Komposition von Blattknoten gänzlich zu verbieten oder Kompositionsregeln für diese anzugeben [4]. Das Beispiel der Merkmale *Login* und *Lokale Anmeldung* zeigt, dass Interaktionen zwischen den Merkmalen in der Praxis benötigt werden, so dass ein Verbot der Verschmelzung von Blattknoten zu restriktiv ist und nachfolgend Kompositionsregeln angegeben werden.

Kompositionsregeln zur Verschmelzung von Attributen. Im untersuchten Ausschnitt des relationalen Datenmodells befinden sich auf Blattebene die Attribute einer Relation. Sie enthalten den Wertebereich und die Definition ob ein Attribut Teil des Primärschlüssels der Relation ist. Die Kompositionsregel für Primärschlüssel besagt, sobald das Attribut in mindestens einem der beiden Merkmale als Schlüssel deklariert wurde, bleibt es Bestandteil des Primärschlüssels der Relation. Durch den von uns in 4.1 vorgestellten Zerlegungsalgorithmus eines globalen Schemas können keine Probleme durch die Erweiterung des Primärschlüssels auftreten. Wird das Schema von Beginn an als variables Schema entworfen, kann es vorkommen, dass durch die Erweiterung des Schlüssels durch ein Merkmal in einem anderen beispielsweise ein *INSERT* Statement nicht mehr funktionieren, weil es keine Kenntnis über das zusätzliche Schlüsselattribut hat und daher einen NULL-Eintrag für dieses erzeugt. In solchen Fällen wird empfohlen Surrogatschlüssel zu verwenden. Ist dies nicht sinnvoll, muss das Merkmal welches den Schlüssel erweitert die DB-Zugriffe aller weiteren Merkmale entsprechend der neuen Primärschlüsseldefinition anpassen. Bei der Komposition von Wertebereichen werden zwei Fälle unterschieden.

Fall 1: Attribute mit identischem Wertebereich. Im ersten Fall wird der Wertebereich in das Ergebnis übernommen. Somit ist die Implementierung des Wertebereichs aller Knoten identisch. Dieser Fall tritt in Abb. 5 auf. In beiden Merkmalen ist für das Attribut *loginname* der Wertebereich *varchar(255)* angegeben und wird daher in das Ergebnis übernommen.

Fall 2: Attribute mit unterschiedlichem Wertebereich. Für den Fall, dass die Wertebereiche unterschiedlich sind, wird davon ausgegangen, dass es sich hierbei um einen Fehler handelt. Eine Superimposition der beiden Merkmale in Abb. 6 ist beispielsweise nicht möglich, da für das Attribut *loginname* unterschiedliche Wertebereiche angegeben sind. Alternativ könnte eine Regel angegeben werden, so dass eine der beiden Modellierungen übernommen wird indem sie eine vorherige überschreibt. Die Erfahrungen zeigen je-

doch, dass es sich hierbei (bisher) immer um eine Inkonsistenz der Modellierung eines Merkmals gehandelt hat, die behoben werden musste.

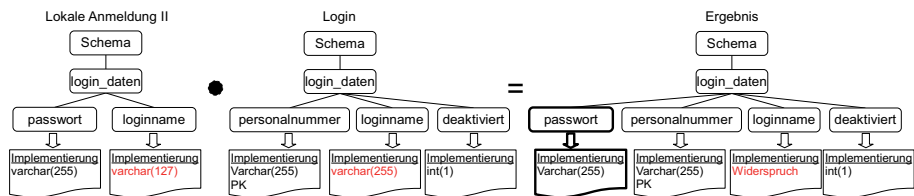


Abbildung 6. Beispiel für Merkmale mit widersprüchlichen Definitionen

6 Anwendung des Ansatzes auf das DB-Schema der Fallstudie ViT

Wir haben bisher gezeigt, wie mittels Superimposition aus den verschiedenen Teilschemata der Merkmale ein Variantschema generiert wird und welche Probleme dabei auftreten können. In diesem Abschnitt wird das Vorgehen an einer realen, nicht trivialen Fallstudie angewendet, um zu ermitteln welche Vorteile das Verfahren mit sich bringt und in welchem Umfang Probleme durch Interaktionen der Merkmale in der Fallstudie auftreten.

6.1 Probleme des globalen Schemas in der Fallstudie

Laut der Befragung des METOP ViT-Entwicklerteams ergeben sich aus der Verwendung eines globalen Schemas in den einzelnen Varianten des ViT-Managers folgende Probleme, die durch die Verwendung unseres Verfahrens gelöst werden sollen:

Umständliche Wartung einer Variante. Die Wartung der produktiv eingesetzten Varianten wird im First-Level vom Kunden selbst übernommen. Die Auslieferung des globalen Schemas erschwert die Verständlichkeit desselben. Falls Daten beispielsweise versehentlich gelöscht wurden, muss zur Wiederherstellung aller Daten in einem konsistenten Zustand das komplette DB-Schema verstanden werden. Momentan ist dies ein komplexer Prozess, da nicht benötigte Schemaelemente vorhanden sind und in dem historisch gewachsenen Schema nicht sofort ersichtlich ist, auf welche Relationen sich die gesuchten Daten verteilen.

Aufwendige Weiterentwicklung der Software. Die Weiterentwicklung der Software erzeugt Probleme, da die Identifizierung welche Merkmale ein Schemaelement benötigen äußerst aufwendig ist. Wird ein Schemaelement während der Weiterentwicklung der Software modifiziert, kann dies Seiteneffekte auf den Anwendungsteil anderer Merkmale ausüben. Es mussten bisher für alle Varianten der Anwendung überprüft werden, inwiefern sie mit dem globalen Schema nach dessen Modifikation kompatibel waren.

Inkonsistente Daten. Erste Versuche Teile des Schemas aus Varianten zu entfernen, führten zu Problemen mit Integritätsbedingungen. In einigen Varianten konnten in einzelnen Relationen keine Daten eingefügt werden, weil der entsprechende referenzierte Datensatz nicht vorhanden war oder Null-Werte für Attribute übertragen wurden, auf denen Not-Null Bedingungen definiert waren. Das Löschen dieser Integritätsbedingung im globalen Schema führte anschließend zu inkonsistenten Daten, die zuvor einen Abbruch der Transaktion erzeugt hätten.

6.2 Das variable DB-Schema

Das monolithische DB-Schema des ViT-Managers wurde, wie in 4.1 beschrieben in Merkmale zerlegt, so dass jedes Merkmal die von ihm benötigten Schemaelemente enthält. In der Tabelle 1 wird die Größe der Modellierung der einzelnen Merkmale angegeben. Die Modellierung des *ViT-Merkmals* enthält beispielsweise 88 Attribute, die sich auf 16 Relationen verteilen. Die Syntaxdarstellung des *ViT-Merkmals* (siehe 4.2) benötigt 135 Zeilen Quellcode.

Tabelle 1. Größe der Merkmalmodellierung

Merkmal	Attribute	Relationen	Codezeilen
ViT	88	16	135
ViT-SPO	80	14	121
BVW	75	12	110
ViT Quadrat	65	12	100
ViT Basic	63	12	98
Terminplanung	35	7	61
...			
LDAP-Login	1	1	3

Aus der Modellierung können maßgeschneiderte Variantenschemata für die einzelnen Varianten des ViT-Managers erstellt werden. In der Abb. 7 sind die Größen der Variantenschemata visualisiert. In der Maximalkonfiguration, in der alle zur Verfügung stehenden optionalen Merkmale ausgewählt wurden, verfügt das DB-Schema über 309 Attribute. Wird das Merkmal *BVW* nicht gewählt, sind im Variantenschema 234 Attribute vorhanden, wird zusätzlich zum *BVW* *ViT-SPO* nicht gewählt 210 usw. Sind die Merkmale *BVW*, *ViT-SPO*, *ViT Quadrat*, *ViT Basic* und *Terminplanung* nicht in der Variante enthalten, wird die Anzahl der Attribute nahezu halbiert. In der Minimalkonfiguration, in der keines der optionalen Merkmale vorhanden ist, sind 146 Attribute enthalten.

6.3 Evaluierung der Ergebnisse der Zerlegung des DB-Schemas

Bei Anwendung des Ansatzes an der Fallstudie stehen Verbesserungen im Bereich Wartung von Varianten und Weiterentwicklung der Software im Vordergrund. Weiterhin soll die Existenz inkonsistenter Daten in Varianten verhindert werden. Eine quantitative Analyse, etwa die Messung der durchschnittlichen Bearbeitungszeit der Wartungsaufträge, ist aufgrund der unterschiedlichen Komplexität der einzelnen Aufträge derzeit

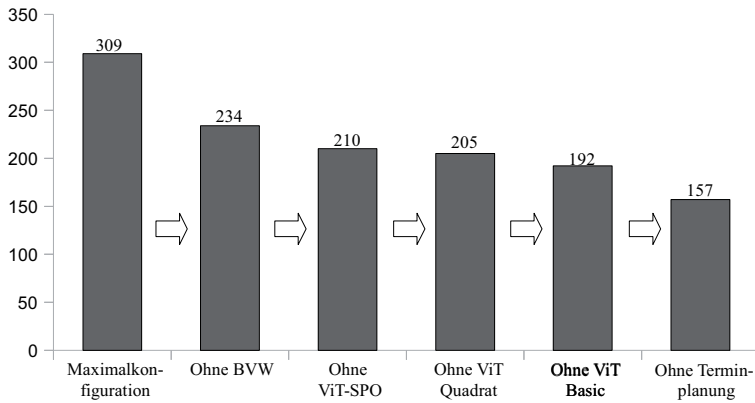


Abbildung 7. Anzahl der Attribute in den Variantenschemata

nicht möglich. Wir nutzen erneut Erkenntnisse aus einer Befragung des METOP ViT-Entwicklerteams.

Wartung einer Variante. Die Wartbarkeit von Variantenschemata konnte deutlich verbessert werden, indem die Komplexität der Variantenschemata reduziert wurde. In Abb. 7 wird veranschaulicht, dass die Größe des Variantenschemas je nach Merkmalauswahl halbiert werden kann. Zusätzlich ist bekannt welche Schemaelemente ein Merkmal verwenden. Somit können z.B. die betroffenen Relationen eines versehentlich gelöschten *MAP-Datensatzes* durch den First-Level Support beim Kunden schneller aufgefunden und durch Umsetzen des Löschröschbits wiederhergestellt werden.

Weiterentwicklung der Software. Die Weiterentwicklung der Software wurde ebenfalls vereinfacht. Das optionale Merkmal zur *Archivierung* der *MAP-Daten* wurde nach der Zerlegung des DB-Schemas in Merkmale implementiert. Durch die Zerlegung war bekannt, welche Daten der *MAPs* archiviert werden müssen, so dass die Implementierung des DB-Schemas des Merkmals und insbesondere der Derivatives, die die zusätzlich benötigten Schemaelemente enthalten, spürbar vereinfacht wurde. Ohne die Zerlegung des Schemas in Merkmale hätten die entsprechenden *MAP* Schemaelemente für jedes Derivative mühsam im Quellcode des *MAP-Merkmals* identifiziert werden müssen. Ähnliche positive Effekte wurden bei der Implementierung des *Daten-Exports* beobachtet.

Inkonsistente Daten. Auf die Wiedereinführung der zuvor gelöschten Integritätsbedingungen wurde bislang verzichtet. Jedoch ist geplant zukünftig Integritätsbedingungen, wie Fremdschlüssel, Not-Null, etc. in die Betrachtung mit einzubeziehen. Wir erhoffen uns ähnlich gute Ergebnisse wie bei den anderen beiden Punkten.

6.4 Interaktionen der Merkmale

Die nach 4.1 erzeugte Zerlegung des monolithischen DB-Schemas erlaubt die Erstellung von Variantenschemata. Nachfolgend wird untersucht in welchem Grad Interaktionen

unter den Merkmalen auftreten. Somit ist es möglich den Einfluss des Problems beispielsweise in Bezug auf Konsistenz und Verständlichkeit der Modellierung besser einschätzen zu können.

Interaktionen durch redundante Schemaelemente. Redundanzen existieren ausschließlich auf Attributebene, da sie sich bei der Superimposition zweier MSB auf der Blattebene befinden und dieses Problem ausschließlich auf Blattebene auftritt (siehe 5.2). An dieser Stelle wird untersucht ob redundant definierte Attribute existieren und für jedes festgehalten, wie oft es definiert wurde. Die Ergebnisse der Analyse sind in Abb. 8 visualisiert. Es sind insgesamt 309 verschiedene Attribute vorhanden. Die Merkmale enthalten jedoch 550 Attributdefinitionen, somit existieren 241 redundante Definitionen, dies entspricht nahezu 44 Prozent der Gesamtmodellierung. Einige Schemaelemente sind in bis zu sechs verschiedenen Merkmaldefinitionen vorhanden. Die Ergebnisse bestätigen, dass das Problem der redundant definierten Schemaelemente in der Fallstudie und somit in der Praxis auftritt. Es erzeugt in der Fallstudie die folgenden Probleme.

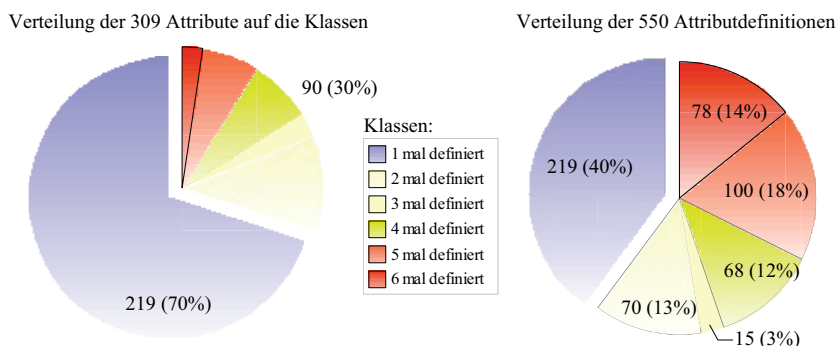


Abbildung 8. Interaktionen durch redundant definierte Schemaelemente

Unübersichtliche Größe der Modellierung. Die redundant definierten Schemaelemente, die 44 Prozent aller Attributdefinitionen ausmachen, erhöhen sowohl die Größe der gesamten Modellierung, als auch die Modellierung der einzelnen Merkmale und lassen diese schnell unübersichtlich werden. In der Fallstudie entstehen Merkmale mit knapp 90 Attributen (siehe Tabelle 1). Dies entspricht fast einem Drittel aller im DB-Schema vorhandener Attribute. Somit wird die Verständlichkeit der Merkmalschemata negativ beeinflusst.

Wahrung der Konsistenz der Modellierung. Es muss sichergestellt werden, dass jedes semantisch identische Attribut den selben Wertebereich erhält. Aufgrund der Generierung der Teilschemata aus einem globalen Schema sind momentan keine widersprüchlichen

Attributdefinitionen vorhanden. Bei der Modifikation eines Schemaelementes während der Weiterentwicklung muss jedoch darauf geachtet werden, dass jede der bis zu sechs redundanten Definitionen des Elementes angepasst wird. Andernfalls entstehen widersprüchliche Definitionen, die die Generierung eines validen Variantenschemas verhindern (siehe 5.3).

Interaktionen durch zusätzlich benötigte Schemaelemente. Das Problem (siehe 4.3), dass aufgrund der Merkmalauswahl zusätzliche Schemaelemente benötigt werden, tritt in der Fallstudie zwischen dem Merkmal *Archivierung* und den *MAPs* sowie zwischen den *MAPs* und dem Datenexport auf. In der Abbildung 9(a) wird gezeigt, dass das Merkmal *Archivierung* je nachdem welche *MAPs* gewählt wurden zusätzliche Schemaelemente zum Archivieren der MAP-Daten benötigt. Hierbei handelt es sich um eine Kopie der MAP-Relationen, in denen die Daten dauerhaft gespeichert werden, um z.B. die Größe der operativen Relation zu verringern. Die zusätzlich benötigten Schemaelemente werden je nach Merkmalauswahl der Basismodellierung durch *Derivatives* hinzugefügt (Abb. 9(b)) in denen die zusätzlich benötigten Elemente enthalten sind. Der Abbildung 9(a) ist weiterhin zu entnehmen, dass das Problem der redundanten Definition ebenfalls an den Überschneidungen der Zusätze, die durch rote Schraffierung gekennzeichnet sind, existiert. Es muss daher ebenfalls sichergestellt werden, dass keine widersprüchlichen Definitionen in den *Derivatives* vorhanden sind.

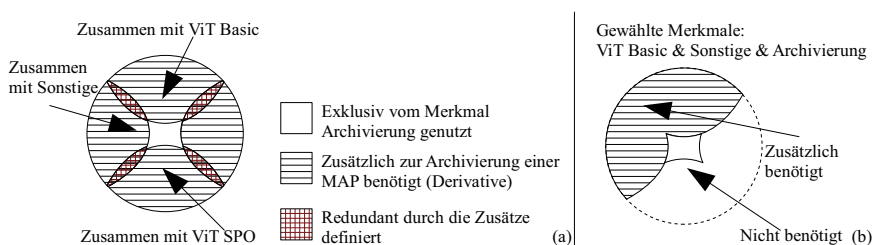


Abbildung 9. Modellierung des Merkmals Archivierung

Insgesamt konnten wir feststellen, dass sich unser Ansatz gut zur Erstellung von maßgeschneiderten DB-Schemata für die Fallstudie eignet. Die Probleme durch Interaktionen der Merkmale sind beherrschbar und es wurden deutliche positive Effekte in den Bereichen *Wartbarkeit* und *Weiterentwicklung* der Software erzielt.

7 Verwandte Arbeiten

Zur Generierung variabler DB-Schemata existieren bislang wenige Arbeiten. In seiner Dissertation [18] verwendet MAHNKE einen komponenten-basierten Ansatz aus dem Variantenschemata zusammengesetzt werden. Er konzentriert sich jedoch auf das objekt-relationale Datenmodell, schließt Interaktionen explizit aus und schlägt zur Anwendung

seines Ansatzes ABLE-SQL eine Spracherweiterung des SQL:1999 Standards [1] vor. Eine solche Modifikation des Standards oder des relationalen Datenmodells [10] wird von dem hier vorgestellten Ansatz nicht benötigt. SIEGMUND et al. präsentieren in [26] eine Anwendung der virtuellen und physischen Trennung der Belange auf das ER-Modell [8]. Wir gehen jedoch einen Schritt weiter und konzentrieren uns auf das relationale Modell. Dies erlaubt die Verwendung des Ansatzes in realen Fallstudien, was wir ebenfalls in dieser Arbeit gezeigt haben. Zusätzliche konnten wir erstmals Aussagen darüber treffen, welche Probleme bei der Generierung maßgeschneiderter DB-Schemata durch Interaktionen der Merkmale auftreten. Weitere Arbeiten zur Anpassung eines DB-Schemas an spezielle Nutzeranforderungen basieren auf Sichten eines globalen DB-Schemas. Dazu gehören Arbeiten zum Thema Sichtintegration [5,22,27] und zur Generierung maßgeschneiderter Sichten [7]. Bei sicht-basierten Ansätzen wird jedoch immer das gesamte Schema ausgeliefert und die Komplexität desselben lediglich vor dem Nutzer durch zusätzliche Sichten versteckt. Die zusätzlichen Sichten erhöhen somit sogar die Komplexität des Schemas, anstatt die Komplexität durch Entfernen der nicht benötigten Elemente zu reduzieren. Weiterhin ist die Erstellung eines Gesamtschemas ausschließlich dann möglich, wenn keine alternativen Merkmale existieren. Mit der Komposition von individuellen Softwarelösungen aus physisch getrennten Fragmenten beschäftigen sich ebenfalls HERMANN et al. [12] und SABETZADEH et al. [23], jedoch liegt ihr Fokus auf der abstrakten Ebene und sie betrachten nicht das relationale Datenmodell oder spezifische Probleme das DB-Schema betreffend.

8 Zusammenfassung

Wir haben gezeigt, wie variable DB-Schemata für das Relationenmodell im Kontext von SPLs modelliert werden können. Für jedes Merkmal wird festgehalten, welche Schemaelemente es benötigt. Somit muss nicht für jede Variante der Software ein DB-Schema modelliert werden, sondern lediglich für jedes Merkmal. Die Modellierung enthält einen Ausschnitt der DDL des SQL:1999 Standards. Es wurde hierfür ein Vorgehen angegeben, mit dem ein zuvor verwendetes globales Schema in Merkmale zerlegt werden kann. Die Generierung eines Variantenschemas erfolgt mittels Superimposition, einem sprachunabhängigen Kompositionsalgorithmus. Hierbei fügen die gewählten Merkmale dem Variantenschema alle von ihnen benötigten Schemaelemente hinzu.

Ziel der Anwendung unseres Ansatzes an der Fallstudie war es zu überprüfen, welche Verbesserungen sich in den Bereichen Wartung, Weiterentwicklung der Software und Vermeidung von Inkonsistenzen ergeben. Durch die Verwendung unseres Ansatzes ließen sich deutliche Verbesserungen im Bereich Wartbarkeit und Vereinfachung der Weiterentwicklung der Software erzielen (siehe 6.3). Somit wurden zwei der drei Ziele erfüllt. Es wurden weiterhin vorbereitende Maßnahmen getroffen, die die Wiedereinführung zuvor entfernter Integritätsbedingungen erlauben. Die Wiedereinführung ist momentan nicht möglich, da wir die entsprechenden Schemaelemente in dieser ersten Arbeit zur Erstellung von Variantenschemata für das relationale Datenmodell nicht betrachtet haben (vgl. 3.2). Hier zeigen sich Verbesserungsmöglichkeiten unseres Ansatzes indem zukünftig weitere Schemaelemente integriert werden müssen.

Weitere Probleme entstehen durch die Verwendung von `SELECT *` und `INSERT INTO TABLE (SELECT ...)` Anweisungen, bei denen die Reihenfolge der Attribute in der Relation von Bedeutung ist (vgl. 4.1). Wir haben die entsprechenden Vorkommen durch die Attribute der in der `FROM` Klausel genutzten Relationen ersetzt. Somit konnten diese Probleme für die Fallstudie gelöst werden, jedoch mag ein solches Vorgehen in weiteren Fallstudien nicht in jedem Fall realisierbar sein. Probleme entstehen weiterhin, wenn Merkmale den Primärschlüssel einer Relation durch zusätzliche Attribute erweitern (vgl. 5.3). Dies kann dazu führen, dass `INSERT` Statements, die z.B. einen `NULL` Wert für das neue Schlüsselattribut liefern, keine Daten in die Relation einfügen können. Durch unseren Zerlegungsalgorithmus existieren solche Vorkommen nicht, da die Merkmale aus einem globalen Schema erzeugt werden. Wird ein Schema jedoch von Anfang an variabel entworfen, ist ein solcher Fall durchaus denkbar.

Insgesamt konnten gute Ergebnisse vor allem in den Bereichen Wartung und Weiterentwicklung, selbst mit der momentan begrenzten Zahl an unterstützten Schemaelementen, erzielt werden. Somit zeigt sich, welches Potential der von uns vorgestellte Ansatz besitzt. Die Probleme des Ansatzes konnten jedoch nicht vollständig gelöst werden, so dass weiterer Forschungsbedarf auf diesem Gebiet besteht. Weiterhin müssen zusätzliche Schemaelemente in die Betrachtung mit aufgenommen und die Ergebnisse an weiteren Fallstudien verifiziert werden.

9 Danksagung

Teile dieser Veröffentlichung entstanden aus dem Forschungsvorhaben „Digitale Fingerspuren (Digi-Dak)“ mit der Projektnummer FKZ:13N10817, welches vom Bundesministerium für Bildung und Forschung (BMBF) gefördert wird. Norbert Siegmund wird unterstützt durch das Bundesministerium für Bildung und Forschung (BMBF), Projektnummer 01IM08003C. Die Arbeit ist Teil des ViERforES Projekts. Christian Kästners Arbeit wird durch den Europäischen Forschungsrat (ScalPL #203099) unterstützt. Abschließend danken wir dem METOP ViT-Entwicklerteam.

Literatur

1. ANSI/ISO/IEC 9075:1999: International Standard - Database Language SQL (1999)
2. Apel, S., Kästner, C., Lengauer, C.: Vergleich und Integration von Komposition und Annotation zur Implementierung von Produktlinien. In: *Software Engineering 2009. Lecture Notes in Informatics*, vol. P-143, pp. 101–112. Gesellschaft für Informatik (GI) (2009)
3. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. In: *Proc. Int'l Softw. Compos. Symp.* pp. 20–35. Springer (2008)
4. Apel, S., Lengauer, C., Möller, B., Kästner, C.: An Algebra for Features and Feature Composition. In: *Proc. Int'l Conf. on Algebraic Methodology and Software Technology. LNCS*, vol. 5140, pp. 36–50. Springer (2008)
5. Batini, C., Lenzerini, M., Navathe, S.: A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys* 18, pp. 323–364 (1986)
6. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30(6), pp. 355–371 (2004)

7. Bolchini, C., Quintarelli, E., Rossato, R.: Relational data tailoring through view composition. In: Proc. Int'l Conf. on Conceptual Modeling. LNCS, vol. 4801, pp. 149–164. Springer (2007)
8. Chen, P.: The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems* 1(1), pp. 9–36 (1976)
9. Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley (2001)
10. Codd, E.: A relational model of data for large shared data banks. *Communications of the ACM* 13(6), pp. 377–387 (1970)
11. Czarnecki, K., Eisenecker, U.: *Generative programming: methods, tools, and applications*. Addison-Wesley (2000)
12. Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: An algebraic view on the semantics of model composition. In: Proc. Europ. Conf. on Model driven architecture-foundations and applications. pp. 99–113. Springer (2007)
13. Johnson, R., Foote, B.: Designing Reusable Classes. *Journal of Object-Oriented Programming* 1(5), pp. 22–35 (1988)
14. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
15. Kästner, C., Apel, S., ur Rahman, S., Rosenmüller, M., Batory, D., Saake, G.: On the impact of the optional feature problem: analysis and case studies. In: Proc. Int'l Conf. on Software Product Line. pp. 181–190. Carnegie Mellon University (2009)
16. Kästner, C., Apel, S.: Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology* 8(6), pp. 59–78 (2009)
17. Liu, J., Batory, D., Lengauer, C.: Feature-Oriented Refactoring of Legacy Applications. In: Proc. Int'l Conf. on Software Engineering. pp. 112–121. ACM (2006)
18. Mahnke, W.: *Komponentenbasierter Schemaentwurf*. Ph.D. thesis, Technische Universität Kaiserslautern, Kaiserslautern, Deutschland (2004)
19. Pohl, K., Böckle, G., Linden, F.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer (2005)
20. Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Proc. Europ. Conf. on Object-Oriented Programming. LNCS, vol. 1241, pp. 419–443. Springer (1997)
21. Saake, G., Heuer, A., Sattler, K.: *Datenbanken Konzepte und Sprachen*. mitp Verlag (2008)
22. Sabetzadeh, M., Easterbrook, S.: View merging in the presence of incompleteness and inconsistency. *Requir. Eng.* 11(3), pp. 174–193 (2006)
23. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M.: Consistency Checking of Conceptual Models via Model Merging. In: Proc. Int'l Conf. on Requirements Engineering. pp. 221–230. Springer (2007)
24. Schäler, M.: *Produktlinientechnologien für den Entwurf variabler DB-Schemata unter Berücksichtigung evolutionärer Änderungen*. Master thesis (diplomarbeit), University of Magdeburg, Germany (2010)
25. Schirmeier, H., Spinczyk, O.: Tailoring Infrastructure Software Product Lines by Static Application Analysis. In: SPLC '07: Proceedings of the 11th International Software Product Line Conference. pp. 255–260. IEEE Computer Society (2007)
26. Siegmund, N., Kästner, C., Rosenmüller, M., Heidenreich, F., Apel, S., Saake, G.: Bridging the Gap between Variability in Client Application and Database Schema. In: Proc. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web. pp. 297–306. Gesellschaft für Informatik (2009)
27. Spaccapietra, S., Parent, C.: View Integration: A Step Forward in Solving Structural Conflicts. *IEEE Trans. on Knowl. and Data Eng.* 6(2), pp. 258–274 (1994)

SIMPL – A Framework for Accessing External Data in Simulation Workflows

Peter Reimann², Michael Reiter¹, Holger Schwarz², Dimka Karastoyanova¹, and Frank Leymann¹

¹ Institute of Architecture of Application Systems, University of Stuttgart
Firstname.Lastname@iaas.uni-stuttgart.de

² Institute of Parallel and Distributed Systems, University of Stuttgart
Firstname.Lastname@ipvs.uni-stuttgart.de

Abstract: Adequate data management and data provisioning are among the most important topics to cope with the information explosion intrinsically associated with simulation applications. Today, data exchange with and between simulation applications is mainly accomplished in a file-style manner. These files show proprietary formats and have to be transformed according to the specific needs of simulation applications. Lots of effort has to be spent to find appropriate data sources and to specify and implement data transformations. In this paper, we present SIMPL – an extensible framework that provides a generic and consolidated abstraction for data management and data provisioning in simulation workflows. We introduce extensions to workflow languages and show how they are used to model the data provisioning for simulation workflows based on data management patterns. Furthermore, we show how the framework supports a uniform access to arbitrary external data in such workflows. This removes the burden from engineers and scientists to specify low-level details of data management for their simulation applications and thus boosts their productivity.

1 Introduction

Workflows have long been used to meet the needs of IT support for business processes. Workflows are compositions of tasks by means of causal or data dependencies that are carried out on a computer using a workflow management system (WfMS) [LR99]. Recently, workflow technology has found application in the area of scientific computing and simulations for implementing complex scientific applications and the term *scientific workflow* has been coined [TDG07]. Simulations, as a subset of scientific applications, are typically compositions of complex calculations and data management tasks, which makes them good candidates for the realization as workflows. For instance, partial differential equations have to be solved to determine temporal or spatial changes of simulated objects, e.g., of the structure of a car in a crash test.

Accessing and provisioning huge amounts of heterogeneous and distributed input data as well as generating huge intermediate and final data sets are some of the major challenges of simulation workflows [TDG07][Gi07][DC08]. Typical data management activities in simulation workflows are extraction, transformation, and load operations (ETL) [Mü10].

In [Vr07], the authors discuss workflow technology as the key technology to cope with heterogeneous applications and data stores. In line with this argumentation and as proposed by [Ma05], our work is based on an ETL workflow approach, i.e., ETL operations of simulation workflows are modeled and executed via workflow technology.

Today, the data management and data provisioning of simulation applications is mainly accomplished in a file-style manner. These files show proprietary formats and inevitably have to be transformed into the appropriate format the simulations require. Most of current scientific workflow management systems (sWfMSs) lack a generic, consolidated, and integrated data management abstraction that can cope with huge and heterogeneous data sets. They use several specialized technologies, e.g., custom workflow activities or services, to access data. Lots of effort must be spent to find appropriate data sources and to specify and implement necessary data transformations, which brings in additional complexity for scientists. This is in particular true for simulations involving multiple domains since each domain has its own requirements and solutions for data handling and thus render the data source and application environment even more heterogeneous. A consolidated abstraction support would remove the burden from engineers and scientists to specify low-level details of data management for their simulation applications.

In this paper, we present SIMPL (SimTech – Information Management, Processes, and Languages) – an extensible framework that addresses the lack of abstraction and generality for data provisioning in current simulation workflow technology. SIMPL provides unified access methods to access arbitrary external data in simulation workflows while metadata describe the mappings between their interfaces and the concrete access mechanisms. At the modeling level, the framework extends the workflow language by a small set of activities that tightly embed data management operations for any kind of data source. When such an activity is executed, it uses the unified access methods of SIMPL to seamlessly access the specified data source. To further assist the workflow modeler in defining typical data management tasks in simulation workflows, we introduce data management patterns, e.g., patterns for ETL operations. In this paper, we show that these patterns in combination with the activities for data management and the unified access methods allow to define the data provisioning for simulations in multiple domains as well as for other scientific applications, such as biology, astronomy, or earthquake science. We discuss the extensibility of the SIMPL framework with respect to additional kinds of data sources and data management patterns. Furthermore, we illustrate the huge potential for a consolidated optimization that SIMPL makes possible as it combines the definition of activities for data management and simulation at the same level of abstraction.

The rest of this paper is organized as follows: Section 2 illustrates the motivation to enhance an existing architecture of sWfMSs by the SIMPL framework and shows its integration into this architecture. Afterwards, Section 3 provides details on major aspects of modeling data management tasks in simulation workflows, while Section 4 deals with the underlying approach to unify heterogeneous access mechanisms for different data sources. We then discuss the benefits and drawbacks of our framework and evaluate it via an example simulation workflow in Section 5. Related work is afterwards discussed in Section 6. Finally, Section 7 concludes and lists future work.

2 The SIMPL Framework

The SIMPL framework is designed as an extension to scientific workflow management systems. Hence, we first sketch the main components of such a system according to the architecture of sWfMSs introduced in [Gö11]. Afterwards, we discuss the motivation to enhance this architecture, illustrate the main aspects by means of a sample workflow, and show the architectural integration of the SIMPL framework.

2.1 Scientific Workflow Management Systems

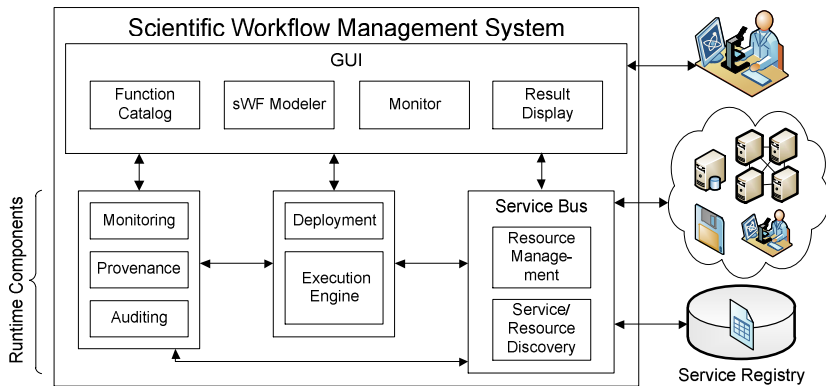


Fig. 1. Architecture of a scientific workflow management system, cf. [Gö11]

The architecture of scientific workflow management systems presented in Figure 1 is based on the workflow technology for business and production workflows as defined in [LR99]. The scientific workflow modeler (*sWF Modeler*) of the GUI supports the modeling of workflow specifications and corresponding deployment information. The *function catalog* provides a list of available services as well as a customizable set of easy-to-model functions that can be used in workflow models. With the help of the *monitor* component, users may constantly observe workflow executions and identify unexpected events or faults. The *result display* component presents the final outcome of simulations as well as intermediate results in a way appropriate for the user.

The *deployment* component transforms workflow models into engine-internal representations and installs them on the *execution engine* that executes instances of these workflows. The *auditing* component records runtime events related to workflows and activities, e.g., the start time of a workflow run. The *monitoring* component uses these events and indicates the states of workflow runs. The *provenance* component records data that goes beyond simple auditing information and that enables the reproducibility of workflow executions. The *service bus* primarily discovers and selects services that implement workflow activities, routes messages, and transforms data. Besides that, it connects workflows to other external, usually stateful resources, e.g., to data sources. The *resource management* component maintains metadata for such external resources as well as for services. The *service/resource discovery* component queries this metadata or external registries to find a list of candidate services or resources by means of descriptive

information, e.g., semantic annotations. This list may be used by the function catalog of the GUI, for late binding of services and resources, or for rebinding of failed activities. This naturally implies the ability to use the modeling tool during the execution of workflow instances to enable ad-hoc changes of workflows [SK10].

In this architecture, scientific workflows may access and handle huge, heterogeneous, and distributed data objects, e.g., via services. However, the challenge still remains to provide a consolidated and integrated data management abstraction that is able to deal with such data objects. This abstraction support is one of the key requirements for scientific workflow management [TDG07][Gi07][DC08]. In the following, we illustrate this challenge using a bone remodeling simulation workflow.

2.2 Simulation Workflow for Bone Remodeling

Figure 2 shows the activities and relevant input and output data of a workflow for a bone remodeling simulation (BRS) that is used to research skeletal disorders, e.g., of human femur. The PANDAS framework calculates the structure of a bone under a specific load using the finite element method (FEM) [KME10]. The workflow is divided into three phases: preprocessing, solving, and post-processing.

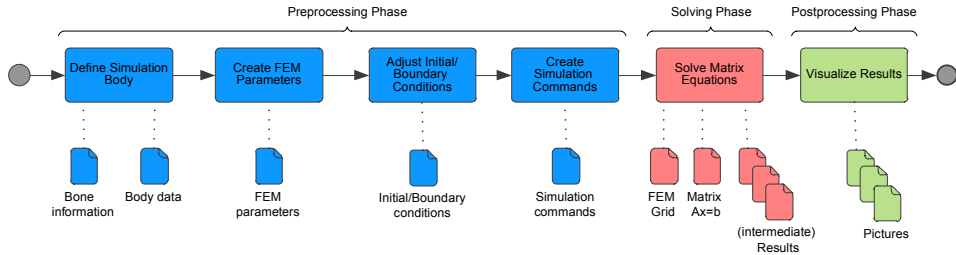


Fig. 2. Workflow for bone remodeling simulation

In the *preprocessing* phase, it starts by loading basic information about the bone to be simulated from different databases or file systems. Examples of this information are a bone structure and material parameters. The second activity extracts FEM parameters from a file, e.g., interpolation functions. Afterwards, the workflow adjusts initial conditions that configure the bone structure for the start time of the simulation. Furthermore, it defines boundary conditions, e.g., the time-dependent pressures from outside on the upper joint of the bone that correspond to the human way of moving. The last preprocessing activity writes a set of simulation commands to a file. For example, it chooses a matrix solver and defines the discretization of the continuous simulation time into n time steps t_1 to t_n . In practice, a simulation involves thousands of such time steps.

In the *solving* phase, the workflow uses the input to create and solve matrix equations for generating the intermediate and final results of the simulation. For each time step t_i , it creates an FEM grid that is the basis to set up matrix equations $\mathbf{Ax} = \mathbf{b}$ that are then solved. The FEM grid contains thousands or millions of mesh points and their relations. This mesh information is typically stored in main memory, but may also be persisted into files or databases for further usage in the post-processing phase. The latter also

holds for the matrix \mathbf{A} and the vectors \mathbf{x} and \mathbf{b} . The solving phase ends after time step t_n . The workflow then stores intermediate and final results based on the vectors \mathbf{x} in comma separated value (CSV) files. The *post-processing* phase transforms these CSV files into another file format suitable for visualization tools.

Altogether, the workflow carries out a multiplicity of data management and data provisioning activities. These activities involve several huge data sets as well as heterogeneous data sources and data formats, e.g., databases, CSV files, unstructured text documents, and image files. Most of the data management operations are performed as manual tasks, implying a high error rate. A generic and consolidated data management abstraction would decrease this error rate. Furthermore, it would remove the burden from scientists to specify low-level details of data management.

2.3 Architecture and main Components of SIMPL

Figure 3 shows how the SIMPL framework extends a sWfMS to provide an abstraction for data management and data provisioning. For better readability, we leave out components of the sWfMS architecture that are not relevant for SIMPL. The *SIMPL core* component, embedded in the service bus, provides unified logical interfaces to any kind of data source. We enhance the resource management component with metadata that describe the mappings between these unified interfaces and the concrete and possibly heterogeneous access mechanisms. The *data management (DM) activity modeling plug-in* of the sWF modeler and the *DM activity execution plug-in* of the execution engine provide data management activities for simulation workflows. These activities may either be directly used in simulation workflows or they may be part of separate ETL workflows that encapsulate data provisioning processes for simulation workflows. The *DM pattern plug-in* of the function catalog assists the workflow modeler in defining the necessary data management operations. It contains abstract data management patterns that allow to model typical data provisioning tasks for simulation workflows. The following sections discuss the SIMPL components and plug-ins in detail.

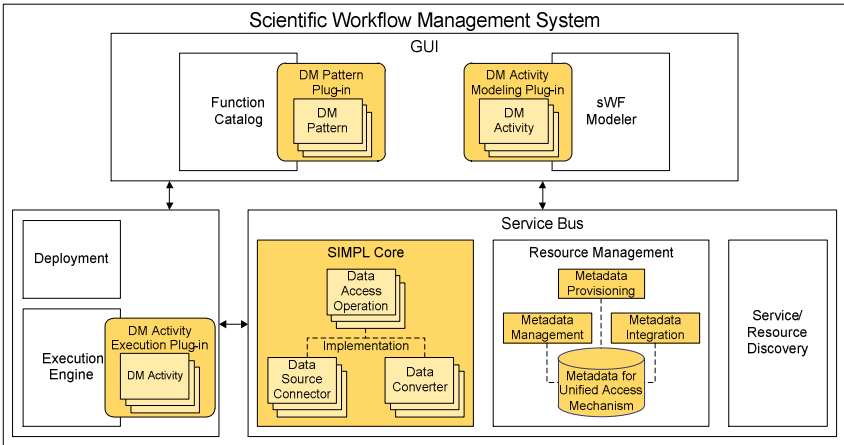


Fig. 3. The SIMPL framework integrated into a sWfMS architecture

3. Modeling Data Management for Simulation Workflows

In this section, we deal with major aspects of modeling data management tasks in simulation workflows. We introduce various extensions to workflow languages that allow for the definition of these tasks. The DM activity modeling plug-in makes these extensions available to the workflow modeler, whereas the DM activity execution plug-in covers their runtime behavior. Furthermore, we show how data management patterns facilitate the definition of data management tasks for simulation workflows.

3.1 Workflow Language Extensions for Data Management

The Business Process Execution Language (BPEL) [Oa07] is the de-facto standard to define and execute business processes based on the control-flow oriented orchestration of service interactions. In [AMA06], BPEL is recommended for modeling and executing scientific workflows and simulation workflows. The main benefits stated are its modular design, its flexibility regarding generic XML data types and late binding of services as well as the fault, compensation, and event handling capabilities. In addition, many BPEL engines offer further capabilities, such as user interaction, workflow monitoring, or recovery of workflows. Due to these benefits of BPEL and in line with previous work, we define the Business Process Execution Language extension for Data Management (BPEL-DM) that extends BPEL by further activity types. We call activities of these new types *data management (DM) activities*. They reflect workflow tasks with embedded data management operations that are seamlessly issued against data sources. The major activity types of BPEL-DM are: *IssueCommand*, *RetrieveData*, and *WriteDataBack*. Each of these activities calls the SIMPL core and sends the data management operation to it in order to deal with heterogeneous data source access mechanisms.

In the following, we use the term *data source* for a system that stores and manages data, e.g., a database or a file system. A data source receives and executes *DM commands*. Examples are SQL statements, shell commands of operating systems, or paths to files. The latter are used to load the content of a file into the process context of the workflow. Each of the DM activities has a BPEL variable as input parameter referring to the data source that executes the embedded DM command. We name such BPEL variables *data source reference variables*. A reference is a *logical data source descriptor* that is either a logical name or a document describing some functional or non-functional requirements for a data source. A logical name describes exactly one data source that is associated with the name in the resource management component. A requirements description can be used for choosing and binding a data source at runtime.

A data source manages several *data containers*. Each container is an identifiable collection of data, e.g., a table in a database system or a file in a file system. *Data container reference variables* refer to a data container via a logical name. The resource management component maps this name to a concrete locator that uniquely identifies the container within the data source. A *data set variable* acts as target container for loading data into the process context of a workflow. Appropriate XML schema definitions specify the contents of these variables and must cope with the differences between

several kinds of data sources. For example, we use an *XML RowSet* structure for any table-oriented data, such as data from an SQL database or from a CSV-based file. XML database systems, as another example, may already provide certain XML schema definitions or they may need to store arbitrary XML data within BPEL variables.

We now detail on the three DM activity types. The *IssueCommand* activity can be used for data manipulation or data definition, for example. Besides the data source reference, it has a DM command as additional input parameter and issues this command against the specified data source. The engine that executes the activity expects a notification whether the DM command has been executed successfully by the data source or not. After a notification of success, the engine continues workflow execution according to the specified control flow. In case of a failure, it enables fault handling mechanisms.

The *RetrieveData* activity also has a DM command as input parameter that is passed to the data source. The DM command must produce data. For instance, it may be a SELECT statement or a path to a file. When the data source has executed the command successfully, the result data is transmitted back to the execution engine. An additional input parameter of the activity defines a data set variable that stores this result data. In case of a failure, the execution engine is notified and enables fault handling mechanisms.

The *WriteDataBack* activity is the counterpart of the *RetrieveData* activity. It writes data from the process context of a workflow back to a data source. The activity accepts one identifier for a data set variable and one for a data container reference variable as input parameters. It stores the data set of the first variable in the data container referred by the second one. As a result, the execution engine gets a notification of success or failure and proceeds like in the case of the *IssueCommand* activity.

Data container reference variables may furthermore be used as parameters in DM commands of the *IssueCommand* or *RetrieveData* activities, e.g., in the FROM clause of an SQL SELECT statement. The same holds for other BPEL variables, e.g., string or integer variables used for comparisons in predicates. The workflow execution engine resolves all these variables, i.e., it reads the variable value and inserts this value at the position within the command where the variable has been referenced beforehand. In order to identify a variable in a DM command and distinguish it from other command items, the variable is marked by surrounding hash marks (e.g., '#'). Regarding a data container reference variable, only the logical name of the data container is inserted in the command. The SIMPL core is later responsible for mapping this name to the data source-specific container identifier by querying the resource management component.

3.2 Abstraction Support through Data Management Patterns

The SIMPL framework provides a set of data management patterns that cover major data provisioning tasks for simulation workflows. The workflow designer picks appropriate patterns from a list provided by the DM pattern plug-in of the function catalog. He/she is then assisted in defining the concrete data management operation for each chosen pattern in a semi-automatic approach instead of defining all details of the operation on his/her own. In the following, we illustrate the pattern-based approach via an example.

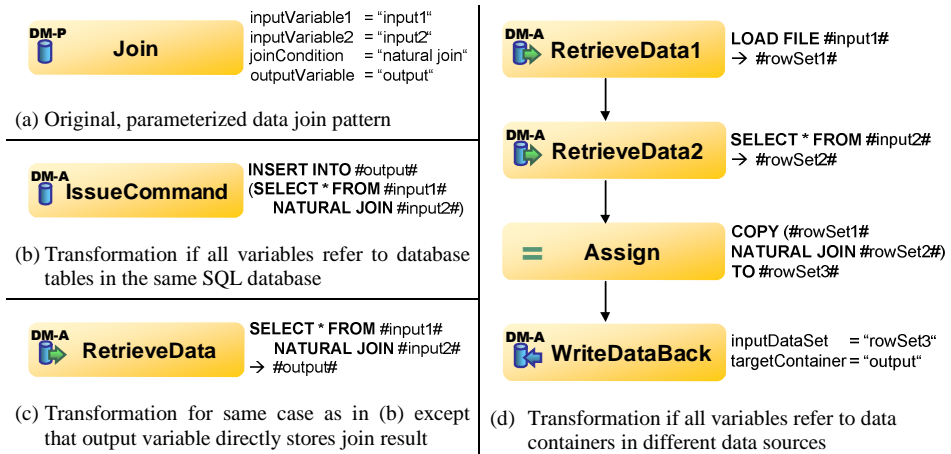


Fig. 4. Data join pattern and its transformation into executable workflow specifications

Figure 4(a) shows a pattern that represents a join of two data sets. Instead of defining concrete data management operations that execute the join, the modeler only needs to set some parameter values, i.e., two input variables, one output variable, and a join condition. Each of the input or output variables may hold data within the process context of the workflow, e.g., via a data set variable of BPEL-DM. Another option is that they refer to external data, e.g., via a data container reference variable. The respective data sets of the two input variables are joined according to the specified join condition. The result of this join is stored in the output variable or in the data container it refers to.

A set of rewrite rules specifies the transformations of abstract and parameterized patterns into workflow parts that carry out the necessary data management operations, e.g., via DM activities of BPEL-DM. The given parameters of the patterns and metadata that describe the characteristics of the data sources to be accessed, e.g., their query capabilities, determine which rewrite rule is to be applied for a certain pattern. Figure 4 shows three rewrite rules for our join example. If the two input variables and the output variable of the join pattern refer to database tables in one and the same SQL database, an *IssueCommand* activity of BPEL-DM with an embedded set-oriented INSERT statement may execute the join (Figure 4(b)). In case the output variable directly stores the join result, we use a *RetrieveData* activity with a SELECT statement (Figure 4(c)). If all three variables refer to data containers in different data sources, the transformation becomes more complex. Assume that we need to perform a join between the content of a CSV-based file and a relational database table and that another database table is the target container for the join result. Then, we may use two *RetrieveData* activities that load the contents of the two input data containers into the process context of the workflow. A subsequent BPEL assign activity joins them, and a *WriteDataBack* activity stores the join result into the target database table (Figure 4(d)).

As described above, metadata about the data sources to be accessed are one basis for deciding on the rewrite rule to be applied for a certain pattern. Hence, we must not apply rewrite rules until it is clear which data sources the data management tasks need to access. In case of a static data source binding during deployment time, we apply rewrite

rules shortly before this deployment phase. If data sources are bound at runtime, we will convert each pattern into a single process fragment and use process fragment technology to dynamically integrate this fragment into an already executing workflow [EUL09].

Besides our join example, the DM pattern plug-in contains further patterns and according rewrite rules for typical data provisioning tasks of simulation workflows. This covers patterns for the transmission of data from one resource to another or for ETL operations [Mü10][TDG07]. ETL operations may be loading or retrieving a bulk of data, filtering a data set as well as joining, merging, or normalizing two data sets.

4 Unified Access Mechanism

Now, we illustrate the approach to unify different kinds of data source access mechanisms. This includes the SIMPL core and its unified logical interfaces to data sources, the metadata to map these interfaces to the underlying access mechanisms, and the interaction between the components of the service bus during data source access.

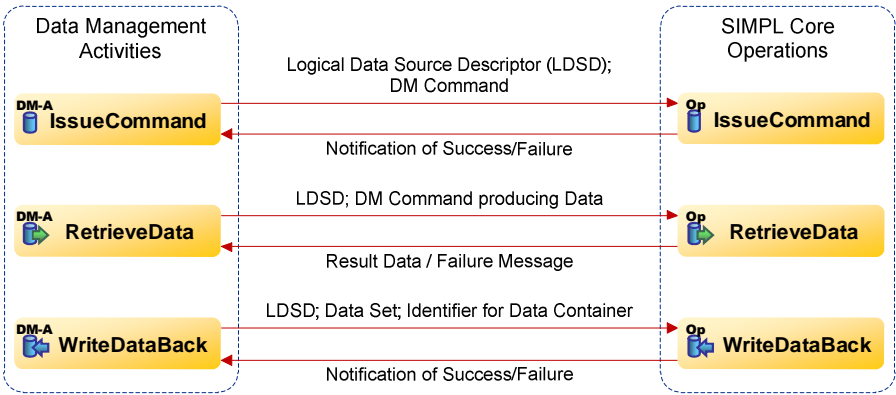


Fig. 5. Data sent between DM activities and SIMPL core operations

4.1 SIMPL Core

The SIMPL core defines a set of generic operations to access arbitrary data sources, i.e., the specifications of the operations are independent of the underlying kinds of data sources. They are geared to the DM activities of BPEL-DM and named accordingly: *IssueCommand*, *RetrieveData*, and *WriteDataBack*. Each DM activity calls the SIMPL core operation that shares the name of the activity. Figure 5 shows which contents of the input parameters of each activity are sent from the workflow execution engine to the corresponding SIMPL core operation and which message or data the activity expects as a reply. Regarding the interaction between workflows and data sources, the SIMPL core operations only forward DM commands, result data, or notifications. They do not implement any complex data transformations or analyses as this would contradict our assumption of workflow activities seamlessly accessing data sources.

Each SIMPL core operation expects a logical data source descriptor as input in order to identify the data source where the data management operation is to be executed. The *IssueCommand* operation gets a DM command as further input, and the *RetrieveData* operation a DM command that produces data. The *WriteDataBack* operation expects a data set and an identifier of the data container to insert the data set, e.g., a logical container name. The *IssueCommand* and *WriteDataBack* operation both deliver a message to the workflow execution engine that indicates whether the data management operation has been successfully executed or not. The *RetrieveData* operation delivers the result data produced by the input DM command in case of success. The workflow execution engine may then store this result data in the data set variable specified for the calling *RetrieveData* activity. In case of failure, the operation delivers a failure message.

Different kinds of data sources rely on different access routines and further properties for data access, e.g., different authentication mechanisms or query capabilities. Hence, the generic access operations of the SIMPL core have to be implemented for concrete data sources or sets of data sources. *Data source connectors* provide this implementation and account for the specific properties of data sources. For example, we use a data source connector for data sources that are based on JDBC access mechanisms. Another connector supports the application programming interface of a certain file system. Some data sources do not support all SIMPL core operations. For instance, sensor nets do not allow for writing data back as they are only able to deliver data. In such a case, the corresponding data source connectors do not provide these operations as well.

The SIMPL core additionally provides *data converters* that transform data from the output format of a data source connector to an XML-based format for the process context of the workflow and vice versa. For instance, a data converter transforms data between the JDBC result set format and the *XML RowSet* format of BPEL-DM. Such data converters may be used for data retrievals or for writing data back to a data source, i.e., for the *RetrieveData* and *WriteDataBack* operations and activities.

4.2 Metadata for Mappings to Heterogeneous Access Mechanisms

We enhance the resource management component of the service bus with metadata about data sources. These metadata describe the mappings between the unified interfaces of the SIMPL core and the underlying and possibly heterogeneous data source access mechanisms. Four kinds of objects may be registered in the resource management component: data sources, data containers, data source connectors, and data converters. Figure 6 shows the classification of the corresponding metadata as well as the cardinalities of associations between individual metadata classes.

A *logical source name* is unique for each data source and acts as its identifier within the SIMPL framework. It can be used as logical data source descriptor within workflows, e.g., in a data source reference variable of BPEL-DM. This constitutes an abstraction offered to the modeler since he/she does not need to deal with real interfaces or security entities. The *interface description* contains information about the interface of the data source, in particular an endpoint to access it. The *security entities*, such as usernames and passwords, enable authorized data source access. The *description of further*

functional or non-functional properties typically includes properties of the data source like the maximally expected response time. Such properties may refer to requirements specified in a logical data source descriptor in order to perform a late binding of data sources. The *data container objects* describe the containers that are managed by the associated data source. They have a *logical container name* assigned that acts as a container reference in workflows, e.g., in a data container reference variable of BPEL-DM. This name is mapped to the concrete *local container identifier* that uniquely identifies the container within the data source.

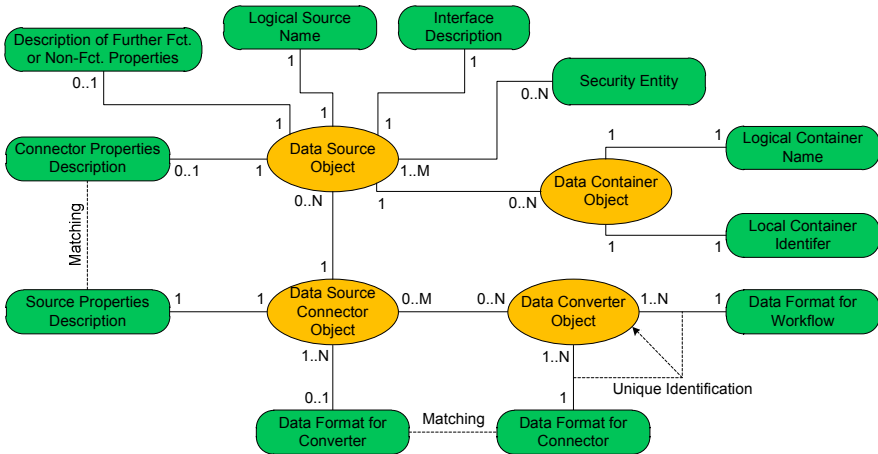


Fig. 6. Classification of metadata to unify heterogeneous data source access mechanisms

As described in Section 4.1, *data source connectors* implement the SIMPL core operations for the data sources they are associated with. Connectors may also be used for multiple data sources, e.g., one connector for all JDBC-based database systems. There might be multiple implementations for a single data source connector registration or data converter registration. In that case, one of these implementations has to be chosen during data source access via additional selection mechanisms, e.g., via the approach of [Ka07]. However, we do not further deal with this aspect for the sake of simplicity.

When a data source is registered or when its registration is updated, the user may directly associate a connector to it. If the user is not sure which connector may handle the data source, he/she may use its *connector properties description*. It describes the properties a connector must have in order to connect to the data source. A similar description, i.e., the *source properties description*, is associated with each data source connector. It describes the properties a connector expects from associated data sources. For instance, both properties descriptions name the SIMPL core operations the associated data source and data source connector support. They are matched to each other to decide on the correct connector for the data source. The same matching can be used when a connector registration is added or updated to find all data sources the connector may handle.

A data source connector is furthermore associated with a description of a *data format for a converter*. It denotes the data format in which the connector delivers output data to a requestor or expects input data from it. A data converter has a similar data format

description associated. These data format descriptions are used to map connectors and converters to each other during the registration of either objects or the update of a registration. So, only those connectors and converters are associated with each other that rely on the same data format. The second data format description associated to a data converter denotes the format in which the converter expects input sent from a workflow and in which it sends its output back to the workflow, e.g., *XML RowSet* of BPEL-DM. The pair of data formats associated with a converter defines between which formats it is able to transform data. As a constraint, this format pair uniquely identifies a data converter object, i.e., there is at most one converter object for each possible pair.

We enhance the resource management component with the functionalities metadata management, metadata provisioning, and metadata integration (see Figure 3). *Metadata management* ensures a persistent and transactional storage of the metadata as well as the management of the metadata schema. The *metadata provisioning* provides metadata information to other components of the sWfMS. It offers a query interface for one or more query languages, e.g., SQL. Besides, it may also offer further repository services that go beyond simple query answering. For example, a service may execute a series of queries that each resolves a selection rule that is used for late binding of data sources. The *metadata integration* is responsible for integrating metadata from internal and external metadata sources and for dealing with according heterogeneities, in particular regarding the metadata schemas and their contents. Such metadata sources can be, e.g., users that access the resource management component via the GUI, external registries that also describe data sources, and the external data sources themselves. Each of these sources may register metadata objects with associated metadata. They may also be asked to complement the metadata after another party has registered an object. For example, a user may register a data source, which then provides all data containers it manages.

4.3 Data Source Access using the SIMPL Framework

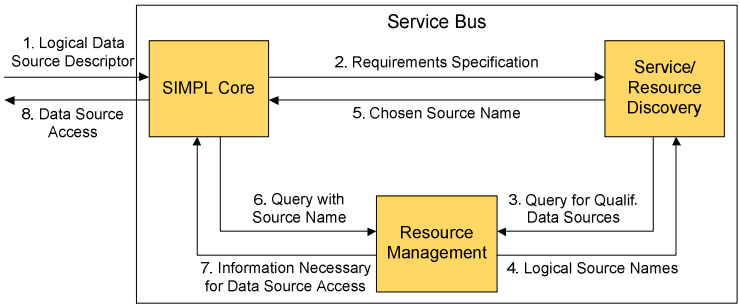


Fig. 7. Interaction of service bus components to prepare data source access

When a workflow accesses a data source via a logical data source descriptor, the SIMPL framework needs to map this descriptor to all information that is necessary for data source access. This information consists of the interface description, a security entity, the suitable data source connector, and the suitable data converter. Furthermore, it needs to map logical names of referenced data containers to local container identifiers. In the following, we describe how the components of the service bus interact with each other to

achieve this mapping. Figure 7 shows this interaction in case the logical data source descriptor sent to the SIMPL core (step 1) contains a requirements specification for a data source. In case it contains a logical name, we skip steps 2 to 5.

The SIMPL core sends the requirements specification to the discovery component (2). The latter queries the resource management component to map the requirements to a set of data source names that identify data sources meeting the claimed requirements (3 and 4). The discovery component then chooses one of these names based on selection criteria in the requirements specification and sends the chosen name back to the SIMPL core (5). The latter queries the resource management component with the source name to retrieve the above-mentioned information that is necessary for data source access (6 and 7). This information is used to access the data source and to execute the SIMPL core operation that is identified by the calling workflow activity (8), e.g., an *IssueCommand* activity calls the *IssueCommand* operation. Strictly speaking, we execute the implementation of this operation as provided by the data source connector identified before.

If the workflow performs a data retrieval or write back, we will need to identify exactly one of the converters that are associated with the identified connector. For that purpose, the workflow engine sends the data type of the BPEL variable that holds the data to be retrieved or written back to the SIMPL core. This data type determines the correct workflow-specific format of the converter. This data format and the connector-specific format assigned to the resolved connector uniquely identify the correct converter.

5 Discussion and Evaluation

In this section, we discuss the benefits and drawbacks of the SIMPL framework. In particular, our discussion covers generality issues of SIMPL, its extensibility, and optimization opportunities for data management in simulation workflows. Afterwards, we evaluate SIMPL via the example workflow for bone remodeling of Section 2.2.

5.1 Generic Data Management for Simulation Workflows

The generic access operations of the SIMPL core, the metadata to describe data sources, and the logical data source descriptors provide a uniform access to arbitrary heterogeneous data sources. This eases the integration of further data management techniques in addition to BPEL-DM. Besides that, we can port the SIMPL core and the metadata management to other sWfMS implementations, which may use different workflow engines, different workflow languages, or even different solutions for modeling data management and data provisioning. The high degree of portability of the framework is basically achieved by its architecture based on clearly separated components and plug-ins extending a sWfMS.

The DM activities of BPEL-DM offer common functionality for data access, data manipulation, data definition, and for writing data back to a data source. Furthermore, SIMPL includes a multitude of data management patterns as further abstraction support.

Together with the uniform data source access and the portability provided by SIMPL, these data management and data access patterns constitute a generic data management solution for simulation workflows. This generality enables SIMPL to be used in multiple domains of simulations or other scientific applications, such as biology and astronomy, and even in the business domain, e.g., for business ETL workflows.

In contrast to our approach, one could provide data services to accomplish the data management for simulation workflows. These services usually offer efficient means for data management functionality specific to a small set of domains or problems. Hence, they do not provide a consolidated, generic, and flexible way to define data management for multi-domain simulation workflows. Nevertheless, since we use BPEL as workflow language, we allow services to be the implementation of data management tasks as well. Furthermore, BPEL processes also offer their functions via service interfaces. So, our approach may even be used to define data services that support special needs of certain domains or problems.

To the best of our knowledge, no other approach includes an abstraction support to define data management operations that is based on generic data management patterns. Typically, workflow modelers have to define low-level details of data management, such as concrete DM commands. By distinguishing the data management operations that are necessary for simulations between different abstract patterns, we can reduce the degrees of freedom in defining the respective operations. This eases the definition and implementation of abstraction mechanisms for individual patterns. Furthermore, the patterns can be seen as building blocks for composing data provisioning workflows, e.g., ETL workflows, via process fragment technology [EUL09]. This increases flexibility at runtime and reduces modeling costs at build time.

5.2 Extensibility of SIMPL

The specifications of the SIMPL core operations and of the DM activities of BPEL-DM are independent of the underlying data sources. The same holds for the logical data source descriptors and the logical data container names. Hence, they do not need to be extended or adapted when SIMPL should support additional kinds of data sources. We only need to add according data sources connectors as well as data converters and XML schema definitions for data set variables. Furthermore, the implementations of connectors and converters as well as the XML schema definitions can typically be derived from already existing implementations or definitions.

In the same sense, we may extend or customize BPEL-DM by additional activities. Like data services, these activities could account for specific needs of a certain scientific domain or problem. To do that, we need to add new SIMPL core operations and their implementations by data source connectors, but only if the already existing operations are not suitable. In order to add a new data management pattern to the DM pattern plugin of the function catalog, suitable rewrite rules have to be defined. These rules describe how the pattern is to be converted into executable workflow parts. Altogether and in contrast to previous approaches, e.g., see the approaches compared in [Vr08], we can typically reuse much of the already existing code for extending SIMPL.

5.3 Optimization of Data Management for Simulation Workflows

Our BPEL-DM approach combines the definition of activities for data management and simulation at the same level of abstraction. This offers a huge potential for a consolidated optimization at both the workflow and the data processing level. In [Vr07], the authors present a flexible approach to optimize workflows with embedded data management operations, in particular SQL statements. Independent of the underlying data sources, this approach shows a huge optimization potential that induces significant performance improvements for workflows. Furthermore, it can be easily applied to other approaches for embedded data management operations, e.g., to our BPEL-DM approach. Due to these optimization options, the SIMPL framework is well suited for a data management and data provisioning abstraction efficiently dealing with huge, heterogeneous, and distributed data objects.

5.4 The Bone Remodeling Workflow in the SIMPL Framework

As a proof of concept, we developed a prototype that implements SIMPL and relevant parts of the associated sWfMS architecture. This prototype uses the Eclipse BPEL Designer¹ as scientific workflow modeler and the Apache Orchestration Director Engine² (ODE) as workflow execution engine and deployment component. Based on the prototype, we implemented the workflow for a bone remodeling simulation (BRS) presented in Section 2.2. Its activities involve several heterogeneous data sources, e.g., databases, CSV files, unstructured documents, or image files. The SIMPL core and the metadata of the resource management provide a uniform access to these data sources.

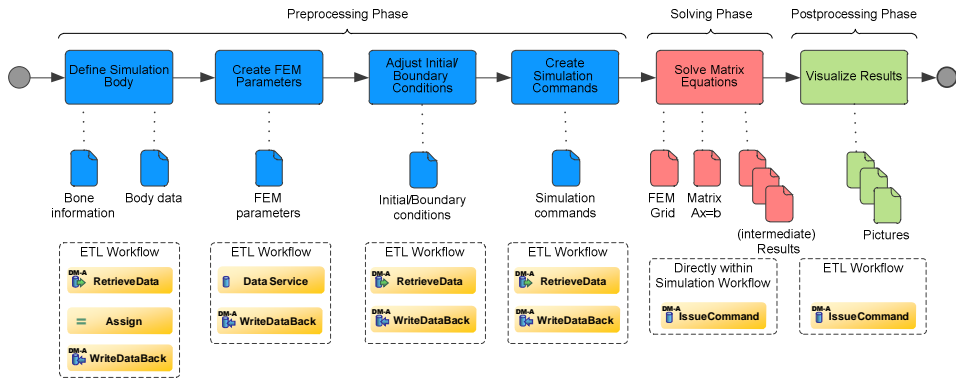


Fig. 8. Workflow for bone remodeling simulation enhanced with SIMPL

Figure 8 shows the BRS workflow using the SIMPL framework. In particular, we show the BPEL-DM activities implementing the main steps of the workflow and its data provisioning at the bottom of the figure. In the preprocessing phase, the workflow creates more than ten input data files each with a size up to one gigabyte. Without the framework, scientists have to select all needed input data, transform the data, and store

¹ <http://www.eclipse.org/bpel/>

² <http://ode.apache.org/>

results into files in the PANDAS environment. SIMPL helps to automate these tasks, thereby reducing the error rate. Input data for the BRS activity *Define Simulation Body*, e.g., bone information or material parameters, are typically stored in public databases or in private file systems. The simulation workflow invokes a separate ETL workflow that converts the data and transfers it into the PANDAS environment. It consists of *RetrieveData* activities that load the input data into BPEL variables. Afterwards, the workflow transforms the data via an assign activity, and *WriteDataBack* activities write the results into the target files. The activities *Adjust Initial/Boundary Conditions* and *Create Simulation Commands* operate according to the same procedure except that they read their input data from structured CSV files. Hence, they also use *RetrieveData* and *WriteDataBack* activities for data selection and transmission.

To perform the FEM, the BRS activity *Create FEM Parameters* has to select certain interpolation functions from an unstructured text document that summarizes all available functions. SIMPL is based on forwarding DM commands to data sources, but the underlying file system does not support executing extractions of unstructured data via DM commands. Hence, SIMPL is not able to directly select these functions. We need data services that select all necessary information and store it into workflow variables. A *WriteDataBack* activity subsequently copies this data into the PANDAS environment.

During the solving phase, the activity *Solve Matrix Equations* calculates matrix equations for several time steps. For each step, it stores all relevant data, i.e., the intermediate results, the FEM grid, and the matrix and vectors into a database inside the PANDAS environment. For selected time steps, a repeatedly executed *IssueCommand* activity in the simulation workflow persists snapshots of these data for further processing, e.g., for analyzing or for recovery purposes. A typical BRS produces 100 or even more of such data snapshots each with a size of about two megabytes. After the last time step, the *IssueCommand* activity stores the final result.

The last BRS activity *Visualize Results* transforms these results and other data, such as the FEM grid, into a format suitable for visualization tools. For example, it joins the FEM grid data and the simulation results for selected time steps of the solving phase in order to create images that combine this information. To automate this, *IssueCommand* activities of an ETL workflow select the necessary data and transform and match it.

The BRS workflow benefits from SIMPL in various ways. SIMPL provides a uniform access to all involved heterogeneous data sources, i.e., databases, CSV files, unstructured text documents, and image files. Furthermore, it allows to automate the data management activities that have previously been performed manually. This reduces the error rate and the time the scientists have to spend for these activities. We chose ETL workflows for the preprocessing and the post-processing phase since they may transfer data directly between the involved data sources. So, we can reduce costs for transmitting high amounts of data and decrease workload for data processing within the simulation workflow. During the solving phase, an *IssueCommand* activity could even transform all FEM data into formats suitable for different solvers, e.g., parallel solvers, i.e., SIMPL helps to switch between these different solvers. Altogether, SIMPL offers a data management abstraction that is well suited for simulation workflows such as the BRS.

6 Related Work

Federated information systems integrate different kinds of data sources and provide a homogeneous schema for heterogeneous source systems [Bu99]. However, they typically involve multiple and sophisticated integration processes that have to be executed for each data source access. In simulation applications the sources are highly heterogeneous and we need to cope with huge amounts of data. Thus, complex integration processes may show poor performance. In that case, a peer-to-peer-based approach seems more suitable as it employs less complex integration processes between pairs of data sources. The generality of our approach recommends it to be used for both a federated and a peer-to-peer-based solution. This also holds for conventional ETL tools. But in contrast to our approach, they rather work with various access mechanisms and data management operators that are specific for a certain kind of data source.

Scientific applications recently adopted the facilities of grid infrastructures as well as the Service Oriented Architecture (SOA) [TDG07]. The most prominent solution for grid- and service-based data management is the Open Grid Services Architecture – Data Access and Integration³ framework (OGSA-DAI). It encapsulates heterogeneous and distributed data sources via services that provide access abstractions for the data sources. A user may define data integration workflows that orchestrate interactions with these services. However, the workflows and workflow tasks of OGSA-DAI are implemented directly in programming languages. If simulation workflows that rely on conventional workflow technology use OGSA-DAI as data management solution, they will not exploit the optimization potential of a consolidated definition of the processing activities for data management and simulation [Vr07]. Furthermore, the abstraction support offered by OGSA-DAI only relies on the customized abstractions offered by the individual services, while we provide a generic and unified abstraction mechanism.

The Scientific Data Management Center (SDM Center) offers an end-to-end data management approach that mainly deals with efficiently analyzing data produced by scientific simulations or experiments [Sh07]. It offers efficient and parallel access routines to storage systems and technologies to support the better understanding of data. The latter comprises, for example, routines for specialized feature discovery, algorithms for parallel statistical data analysis, and efficient indexes over large and distributed data sets. On top of this, the Kepler sWfMS provides the robust automation of processes for generating, collecting, and storing the results of simulations or experiments as well as for data post-processing and analyzing the results [Lu06]. In contrast to the SDM Center, our approach does not deal with data analysis, but with data provisioning for simulation workflows and an appropriate abstraction support. Kepler also offers workflow activities to seamlessly access data sources, in particular for local file systems, relational database systems, and data streams coming from sensor networks. However, each of these activities directly deals with the heterogeneities regarding access mechanisms of the considered data sources instead of using generic and unified interfaces.

³ <http://www.ogsadai.org.uk/index.php>

The scientific workflow management system VisTrails, focuses on the exploration and visualization of results of simulations or experiments as well as on modeling, executing, and optimizing visualization workflows [Fr06]. It supports tracking revisions of workflows, i.e., scientists or engineers may interactively adjust their workflows. In order to maintain the history of workflow execution, data processing, and workflow revisions, VisTrails captures data and workflow provenance and links them to each other and to the produced data [Ko10]. This enables reproducibility of processes and simplifies the exploration of different versions of a workflow as well as its results. In contrast to the framework presented in this paper, VisTrails does not focus on data provisioning aspects and abstractions that are necessary for executing all phases of simulations.

Microsoft Trident is a general-purpose scientific workflow workbench [Ba08]. It is built on top of the Microsoft Windows Workflow Foundation⁴ (Windows WF), a workflow environment based on the control-flow oriented Extensible Orchestration Markup Language (XOML). Trident enhances Windows WF with functionality needed for scientific workflow management, e.g., automatic provenance capture and the possibility to model data dependencies between workflow tasks. The activity library of Windows WF enables customized activity types that could provide a seamless access to data sources or further abstractions for defining data management operations. However, they have to be implemented by the modeler himself or shared between several activity developers. SIMPL offers abstractions via data management patterns that are automatically converted into executable workflow parts. As an alternative to such custom activity types, Trident uses services for data access. Similar to OGSA-DAI, this complicates optimizations over the whole spectrum from the workflow to the data processing level. Furthermore, Trident workflows may use Dryad for data provisioning [Is07]. Following the approach of MapReduce [DG04], Dryad supports programmers in efficiently using multiple resources for executing data-intensive and data-parallel applications without knowing anything about concurrent programming. However, Dryad does not deal with data management abstractions in our sense of a generic solution.

Besides the activity library of Windows WF, IBM and Oracle also provide workflow activities that directly embed data management operations as part of their workflow products [Vr08]. In contrast to SIMPL, these products do not offer abstractions via data management patterns and are restricted to SQL statements, while we support any kind of data source. The external variables of Apache ODE are another approach to seamlessly access data sources from within workflows. These variables can be mapped to one row of a table in a database that offers an interface following Java Database Connectivity⁵ (JDBC). This way, workflows may perform tuple-oriented retrievals and manipulations on the mapped row. However, set-oriented operations have to be defined via additional workflow constructs, e.g., loop activities. In [Vr07], the authors proof that such a loop-based execution of several tuple-oriented operations shows weak performance related to a set-oriented SQL statement that is wholly executed by the database system. Our approach supports set-oriented operations by directly integrating SQL statements into the workflow definition.

⁴ <http://www.windowsworkflowfoundation.eu/>

⁵ <http://java.sun.com/products/jdbc/overview.html>

7 Conclusion and Future Work

In this paper, we introduced SIMPL – an extensible framework that provides a generic and consolidated abstraction for data management and data provisioning in simulation workflows. It unifies heterogeneous interfaces to different data sources via logical data source descriptors, generic access operations, and metadata for mappings to concrete data source access mechanisms. We demonstrated that this provides the core functionality to uniformly access arbitrary data sources and enables an easy development and integration of concrete data management techniques. Based on this, the BPEL-DM activities allow for the definition and execution of common data management and data provisioning tasks for simulation workflows. Further abstraction support is provided by means of generic data management patterns, e.g., patterns for ETL operations. In addition to a data source access via services, BPEL-DM offers the combined definition of the processing activities for data management and simulation at the same level of abstraction. This enables optimizations over the whole spectrum from the workflow level to the data level, inducing significant performance improvements of workflows. Altogether, the SIMPL framework removes the burden from engineers and scientists to specify low-level details of data management for their simulations. It helps them to cope with the information explosion intrinsically associated with simulation applications and boosts their productivity.

In future, we will extend the optimization approach for workflows with embedded data management operations of [Vr07] to be applicable to the data management in simulation workflows. For that purpose, we will work on a set of optimization rules that are suitable for simulation workflows and for DM activities of BPEL-DM. Scientists may use several parameterized data management patterns within their workflows. Our approach converts each pattern into an executable workflow part in isolation from all other patterns. This may result in a variety of process fragments for data management and data provisioning that show further optimization potential when considered together. To exploit this optimization potential, we will combine the conversion of data management patterns with the optimization approach for data management.

Acknowledgement: The authors would like to thank the German Research Foundation (DFG) for financial support of the projects within the Cluster of Excellence in Simulation Technology at the University of Stuttgart.

Bibliography

- [AMA06] Akram, A.; Meredith, D.; Allan, R.: Evaluation of BPEL to Scientific Workflows. In: Proc. of the 6th International Symposium on Cluster Computing and the Grid (CCGRID '06), Singapore, Malaysia, 2006.
- [Ba08] Barga, R. et. al.: The Trident Scientific Workflow Workbench. In: Proc. of the 4th International Conference on e-Science, Indianapolis, Indiana, 2008.
- [Bu99] Busse, S. et. al.: Federated Information Systems: Concepts, Terminology and Architectures. Research Report of the Faculty of Computer Science at the Technische Universität Berlin, 1999.

- [DC08] Deelman, E.; Chervenak, C.: Data Management Challenges of Data-Intensive Scientific Workflows. In: Proc. of the 8th International Symposium on Cluster Computing and the Grid (CCGRID '08), Washington, DC, 2008.
- [DG04] Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of the 6th Symposium on Operating Systems Design and Implementation, San Francisco, California, 2004.
- [EUL09] Eberle, H.; Unger, T.; Leymann, F.: Process Fragments. In: On the Move to Meaningful Internet Systems: OTM 2009, Part I. Springer, 2009.
- [Fr06] Freire, J. et. al.: Managing Rapidly-Evolving Scientific Workflows. In: Proc. of the 1st International Provenance and Annotation Workshop (IPAW), Chicago, Illinois, 2006.
- [Gi07] Gil, Y. et. al.: Examining the Challenges of Scientific Workflows. In: IEEE Computer, vol. 40, no. 12, December 2007.
- [Gö11] Görlach, K. et. al.: Conventional Workflow Technology for Scientific Simulation. To appear in: Yang, Y. (ed.); Wang, L. (ed.); Jie, W. (ed.): Guide to e-Science. Springer, 2011.
- [Is07] Isard, M. et. al.: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In: Proc. of the European Conference on Computer Systems (EuroSys), Lisbon, Portugal, 2007.
- [Ka07] Karastoyanova, D. et. al.: Semantic Service Bus: Architecture and Implementation of a Next Generation Middleware. In: Proc. of the 2nd International ICDE Workshop on Service Engineering (SEIW 2007), Istanbul, Turkey, 2007.
- [KME10] Krause, R.; Markert, B.; Ehlers, W.: A Porous Media Model for the Description of Adaptive Bone Remodelling. In: Proc. of the 81st Annual Meeting of the International Association of Applied Mathematics and Mechanics, Karlsruhe, Germany, 2010.
- [Ko10] Koop, D. et. al.: Bridging Workflow and Data Provenance using Strong Links. In: Proc. of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM), Heidelberg, Germany, 2010.
- [LR99] Leymann, F.; Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall, Englewood Cliffs, NJ, 1999.
- [Lu06] Ludäscher, B. et. al.: Scientific Workflow Management and the Kepler System. In: Concurrency and Computation: Practice and Experience, vol. 18, issue 10, 2006.
- [Ma05] Maier, A. et. al.: On Combining Business Process Integration and ETL Technologies. In: Gesellschaft für Informatik (ed.): Datenbanksysteme in Business, Technologie und Web, Karlsruhe, Germany, 2005.
- [Mü10] Müller, C.: Development of an Integrated Database Architecture for a Runtime Environment for Simulation Workflows. Diploma Thesis, University of Stuttgart, 2010.
- [Oa07] OASIS: Web Services Business Process Execution Language Version 2.0, 2007.
- [Sh07] Shoshani, A. et. al.: SDM Center Technologies for Accelerating Scientific Discoveries. In: Proc. of the Scientific Discovery through Advanced Computing Conference (SciDAC 2007), Boston, Massachusetts, 2007.
- [SK10] Sonntag, M.; Karastoyanova, D.: Next Generation Interactive Scientific Experimenting Based on the Workflow Technology. In: Proc. of the 21st IASTED International Conference on Modelling and Simulation (MS 2010), Prague, Czech Republic, 2010.
- [TDG07] Taylor, I.; Deelman, E.; Gannon, D.: Workflows for e-Science - Scientific Workflows for Grids. Springer, London, UK, 2007.
- [Vr07] Vrhovnik, M. et. al.: An Approach to Optimize Data Processing in Business Processes. In: Proc. of the 33rd International Conference on Very Large Data Bases (VLDB 2007), Vienna, Austria, 2007.
- [Vr08] Vrhovnik, M. et. al.: An Overview of SQL Support in Workflow Products. In: Proc. of the 24th International Conference on Data Engineering (ICDE 2008), Cancún, México, 2008.

Einsatz domänenspezifischer Sprachen zur Migration von Datenbank Anwendungen

Sven Efftinge¹, Sören Frey², Wilhelm Hasselbring², Jan Köhnlein¹

¹itemis AG

Schauenburgerstraße 116, 24118 Kiel

<http://www.itemis.de/>

²Universität Kiel

Institut für Informatik, Software Engineering, 24118 Kiel

<https://se.informatik.uni-kiel.de/>

Abstract: In der modellgetriebenen Softwareentwicklung von Datenbank Anwendungen werden domänenspezifische Sprachen zusammen mit Codegeneratoren eingesetzt, welche aus formalen Modellen automatisiert lauffähige Software erzeugen. Selbstdefinierte domänenspezifische Sprachen werden zusammen mit dazu selbstentwickelten Codegeneratoren eingesetzt, sie können aber auch direkt in Wirtssprachen integriert werden, ohne eine spezielle Werkzeugumgebung zu benötigen. Sogenannte *externe* domänenspezifische Sprachen werden durch einen selbst erstellten Parser verarbeitet, während *interne* domänenspezifische Sprachen in eine Wirtssprache eingebettet werden und somit deren Parser mit nutzen.

Für das Szenario der Migration von Datenbank Anwendungen (von OracleForms zu Java Swing) präsentieren wir den Einsatz externer und interner domänenspezifischer Sprachen für unterschiedliche Aufgaben eines größeren Migrationsprojektes. Wir betrachten die Überlegungen zur Migrationsarchitektur für den konkreten Kontext des Projekts Forms2Java anhand des Dublo-Migrationsmusters. Neben der eigentlichen Vorstellung der Sprachen diskutieren wir auch die generelle Fragestellung, wann sich der mit dem Einsatz domänenspezifischer Sprachen verbundene Aufwand lohnt.

1 Einleitung

Die Stärken eines großen Teils der deutschen Softwareindustrie sind geprägt durch ein tiefes Verständnis der jeweiligen Anwendungsdomänen. Das wohl bekannteste Beispiel dazu ist die SAP mit ihren betriebswirtschaftlichen Anwendungssoftwaresystemen. Es gibt aber auch sehr viele weitere Unternehmen, die erfolgreich Standard- und Individualsoftware für bestimmte Domänen entwickeln. Eine zentrale Frage ist dann, wie diese Stärken in den jeweiligen Anwendungsdomänen effektiv und effizient in der eigentlichen Softwareentwicklung unterstützt werden können. Dazu bieten sich u.a. die modellgetriebene Softwareentwicklung und der damit verbundene Einsatz sogenannter domänenspezifischer Sprachen (englisch: Domain-Specific Languages, DSL) als viel versprechende Ansätze an.

Modellgetriebene Softwareentwicklung ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen [SVE07]. Dabei werden DSLs zusammen mit entsprechenden Codegeneratoren eingesetzt. Im Bereich der modellgetriebenen Entwicklung richten sich DSLs verstärkt auf die Lösung von Problemen in einer

bestimmten Anwendungsdomäne aus. Oft wird ein generativer Ansatz verfolgt, um die Modelle der DSLs durch eine automatische Transformation in ausführbaren Code zu überführen. Im Gegensatz zu sogenannten „General Purpose Languages“ sind DSLs auf ein Anwendungsgebiet optimal zugeschnitten und abstrahieren von der zugrunde liegenden Programmierplattform. Somit können domänenspezifische Sachverhalte viel präziser und knapper formuliert werden.

DSLs können anhand des Einbettungsgrades unterschieden werden: Sprachen, deren eigene Syntax durch einen selbst erstellten Parser verarbeitet wird, nennt man *extern*. Sprachen, die in ihre Wirtsprache eingebettet sind und damit lediglich eine Spezialisierung dieser Wirtsprache darstellen, nennt man *intern*. In diesem Papier stellen wir einen Ansatz zur Migration von Datenbank Anwendungen vor, in dem die folgenden Arten von DSLs eingesetzt werden:

- Eine externe DSL dient zur Datenmodellspezifikation, aus der der Datenbankzugriffscode generiert wird (hier implementiert mit der Java Persistence API).
- Eine interne DSL dient zur Spezifikation der Integritätsbedingungen, die bei der Dateneingabe überprüft werden (hier mit Java als Wirtsprache).
- Eine externe DSL dient zur Spezifikation des Layouts der Benutzungsschnittstellen, aus der Eingabemasken für die Datenbank generiert werden (hier implementiert mit Java Swing und unterstützt durch einen speziellen GUI-Editor).

Dieser Ansatz wurde in einem größeren Projekt erfolgreich eingesetzt. Eine wichtige Frage in diesem Zusammenhang ist es immer, ab wann sich der mit dem Einsatz einer DSL verbundene Aufwand rechnet; dieser Fragestellung werden wir ebenfalls diskutieren.

Im Folgenden werden wir zunächst in Abschnitt 2 auf den Projektkontext mit einigen quantitativen Angaben zum Projektumfang und einer Beschreibung der Migration von Datenbank Anwendungen mit dem Dublo-Migrationsmuster eingehen. Die oben erwähnten internen/externen DSLs werden in den Abschnitten 3 bis 5 vorgestellt. In Abschnitt 6 diskutieren wird die Kosten und das Einsparpotenzial des vorgestellten Ansatzes, Abschnitt 7 stellt verwandte Arbeiten vor, bevor Abschnitt 8 das Papier zusammenfasst und einen Ausblick auf geplante Arbeiten liefert.

2 Projektkontext und Migrationsarchitektur

Im Projekt *Forms2Java* wurde eine Datenbank Anwendung der APG Affichage¹ zur Verwaltung von Geschäfts- und Kundendaten migriert. Das Altsystem *Gepard* ist eine Datenbank Anwendung, die mittels der Oracle-Datenbank und OracleForms für die Entwicklung der Benutzungsschnittstellen programmiert wurde.

Weil die über Jahre gewachsenen OracleForms sich zunehmend schlechter warten ließen und gleichzeitig die Zukunft der OracleForms-Technologie selbst infrage stand, entschloss

¹<http://www.apgsga.ch/>

man sich zu einer Migration auf eine besser skalierende, zukunftssichere Plattform unter Beibehaltung des unternehmenskritischen Datenbestands. Die Altanwendung umfasst 1722 Tabellen mit 19572 Spalten und über 300 Forms.

Es gibt verschiedene Möglichkeiten, ein Altsystem in eine neue Architektur zu migrieren. Altsysteme stellen wichtige Investitionen dar, die nicht einfach außer Betrieb genommen werden können. Der Betrieb muss während des Übergangs weitergehen. Folglich sind sanfte Migrationspfade und die Integration von Alt- und Neusystemen essenziell für die Praxis der Integration von Informationssystemen [BS95, HKRS08].

In [HBG⁺08] wurden auf Basis der Erfahrungen aus mehreren Migrationsprojekten, in denen insbesondere im Bereich der Datenbankintegration immer wiederkehrende Probleme auftraten, drei Varianten des ursprünglichen Dublo-Musters [HRJ⁺04] abgeleitet:

Dublo Service Die Dublo-Service-Lösungsstruktur ist in Abbildung 1(a) dargestellt. Die Grundidee besteht in der Entwicklung der Geschäftslogik in der neuen Geschäftslogikschicht, der Erstellung eines Legacy-Adapters für den Zugriff der neuen Geschäftslogik auf die existierende Legacy-Geschäftslogik und der Benutzung dieses Adapters für den Datenzugriff. Folglich wird auf die Datenbank nur indirekt durch den vorhandenen Legacy-Code zugegriffen.

Dublo Old Database Die Dublo-Old-Database-Lösungsstruktur ist in Abbildung 1(b) dargestellt. Die Grundidee besteht darin, dass im Gegensatz zum Dublo-Service-Muster direkt auf die Legacy-Datenbank zugegriffen wird. Diese Strategie erhält die alte Datenbank und ersetzt die alte Kombination aus Präsentations-, Geschäfts- und Datenzugriffsebene.

Dublo New Database Die Dublo-New-Database-Lösungsstruktur ist in Abbildung 1(c) dargestellt. Die Grundidee besteht darin, parallel zum Altsystem eine ganz neue Infrastruktur aufzubauen.

Die Anwendung des Dublo-Service- und des Dublo-Old-Database-Musters ist sinnvoll, wenn ein inkrementeller Austausch alter Geschäftslogik- und Client-Software durch neue Geschäftslogik in der Mittelschicht angestrebt wird. Da keine zusätzliche Datenbank eingeführt wird, entstehen keine Konsistenz- oder Abgleichsprobleme zwischen neuer und alter Datenbank. Mit dem Dublo-Service-Muster ist es für die neuen Clients transparent, ob Geschäftslogik bereits in der neuen Mittelschicht oder noch im alten Legacy-Code implementiert ist.

In [HBG⁺08] werden Kriterien und Erfolgsfaktoren zum Einsatz der jeweiligen Dublo-Variante angegeben. Für das Projekt *Forms2Java* ist Dublo Old Database die richtige Variante, weil ein Parallelbetrieb von Alt- und Neusystem in einer Übergangszeit erforderlich ist und das existierende Datenbankschema ausreichend gut strukturiert ist. Das Datenbankschema von Gepard wurde stets diszipliniert weiterentwickelt, so dass es auch zukünftig eingesetzt werden kann. Die darauf aufsetzenden Anwendungen wurden nun migriert, von OracleForms zu Java Swing. Ein weiterer Erfolgsfaktor für das Dublo-Old-Database-Muster besteht darin, dass davon ausgegangen werden kann, dass das DBMS vom Hersteller weitergepflegt werden wird.

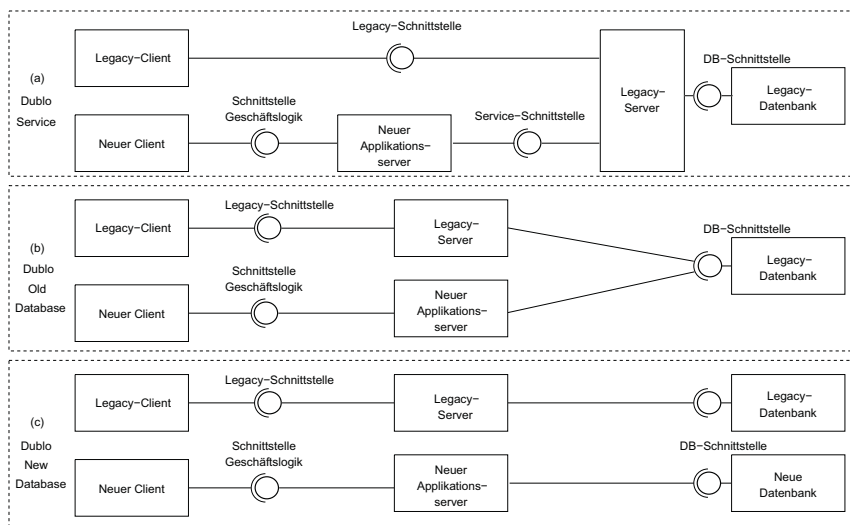


Abbildung 1: Die Varianten des Dublo-Musters [HBG⁺08].

Die Zielarchitektur für das neue *Gepard*-System nutzt, wie das Altsystem, die Oracle-Datenbank, für die Anwendungsprogrammierung werden jedoch die Java Persistence API² und das Spring Framework³ verwendet. Die neuen Benutzungsschnittstellen werden als „Rich Clients“ mit der Programmierschnittstelle und Grafikbibliothek Swing⁴, sowie dem JGoodies Framework⁵ realisiert.

Abbildung 2 zeigt eine Übersicht über die aktualisierte Architektur der Anwendung *Gepard* und die Rollen der eingesetzten DSLs im Konstruktionsprozess. Auf der rechten Seite sind die vier Schichten des Anwendungssystems dargestellt, welches entwickelt werden soll. Die interne DSL für die Eingabeprüfung wird in Abschnitt 4 vorgestellt, bevor die GUI-DSL für das Layout der Eingabemasken in Abschnitt 5 erläutert wird. Im folgenden Abschnitt 3 zeigen wir, wie eine mittels TMF Xtext⁶ implementierte externe DSL für die Beschreibung des Datenmodells verwendet werden kann. Diese externe DSL referenziert aus dem Datenbanksystem exportierte Schema-Informationen und wird mittels XPand aus dem openArchitectureWare-Projekt⁷ transformiert, welches inzwischen in das Eclipse Modeling Projekt überführt wurde.⁸ Die interne Java-DSL wird über eine Java-5-Annotation in die Eingabeprüfung integriert, siehe Abschnitt 4. Für die GUI-DSL wurde ein spezifischer graphischer Editor auf Basis des Graphical Modeling Frameworks⁹ entwickelt, siehe Abschnitt 5.

²<http://www.oracle.com/technetwork/java/javasee/tech/>

³<http://www.springsource.org/>

⁴<http://download.oracle.com/javase/tutorial/uiswing/>

⁵<http://www.jgoodies.com/>

⁶<http://www.xtext.org/>

⁷<http://www.openarchitectureware.org/>

⁸<http://www.eclipse.org/modeling/>

⁹<http://www.eclipse.org/gmf/>

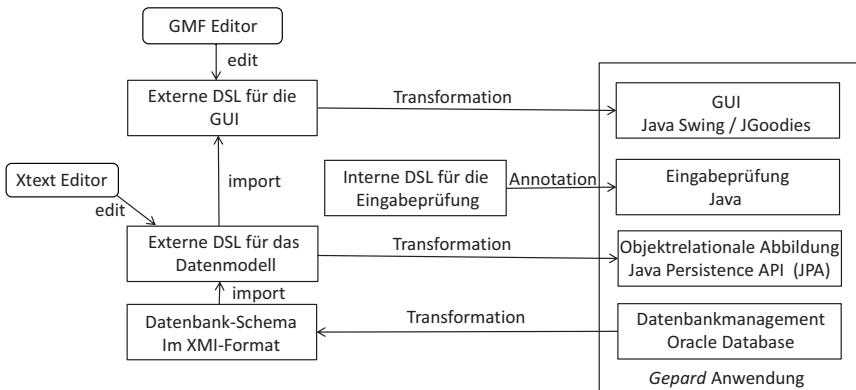


Abbildung 2: Übersicht über die aktualisierte Architektur der Anwendung *Gepard* (rechts) und die Rollen der eingesetzten DSLs im Konstruktionsprozess.

3 Einsatz einer externen DSL für das Datenmodell

Einen kleinen Ausschnitt des Programmcodes für den Zugriff aus Java mit der Java Persistence API (JPA) auf das relationale DBMS zeigt Abbildung 3(a). Die JPA wurde mit Version 5 der Java EE Plattform eingeführt. Sie stellt eine standardisierte Schnittstelle für Java-Anwendungen dar, die die Zuordnung und die Übertragung von Laufzeit-Objekten einer Java-Anwendung in relationale Datenbanken ermöglicht, also eine objektrelationale Abbildung implementiert. Die JPA zeichnet sich durch einen umfangreichen Einsatz von Java 5 Annotationen aus, wie es in Abbildung 3(a) zu sehen ist. Da die Datenbank ohne Änderungen migriert werden soll, können die Voreinstellungen der JPA in vielen Fällen nicht benutzt werden, was die Anzahl der benötigten Annotationen drastisch erhöht. Dieser Code wiederholt sich vom Grundmuster her immer wieder, so dass es bei der Größe der zu migrierenden Anwendung lohnenswert erscheint, diese Zugriffsschicht aus einer Datenmodell-DSL zu generieren.

Eine externe DSL zur Beschreibung des Datenmodells der *Gepard*-Datenbank wurde mittels Xtext realisiert. Ein Beispiel für die Programmierung mit dieser DSL ist in Abbildung 3(b) dargestellt. Generiert wird aus dieser Datenmodell-DSL die objektrelationale Abbildung, wie sie in Abbildung 3(a) zu sehen ist.

Defaults für das Datenmodell werden aus dem Schema der Datenbank extrahiert, siehe Abbildung 2. Um eine gute Referenzierbarkeit des Datenbankschemas aus der Datenmodell-DSL zu ermöglichen, wird die Schemainformation aus der Datenbank in eine XMI-Datei transformiert. XML Metadata Interchange (XMI)¹⁰ ist ein Standard-Austauschformat für Modelle zwischen Software-Entwicklungswerkzeugen, welcher insbesondere für den Austausch von UML-Modellen entwickelt wurde.

Falls die Datenstrukturinformationen aus dem Datenbankschema ausreichen, um die objektrelationale Abbildung zu generieren, genügt die folgende Datenbank-DSL-Spezifikation:

¹⁰<http://www.omg.org/spec/XMI/>

(a)

```
@SuppressWarnings("serial")
@Entity
@Table(name = "BUCHUNGSKREISE_F")
public class BuchungskreiseF
    extends AbstractEntity implements Serializable {

    @SuppressWarnings("unused")
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "bkrIdSeq")
    @SequenceGenerator(name = "bkrIdSeq", sequenceName = "SEQ",
        allocationSize = 1)
    @Column(name = "BKR_ID", nullable = false)
    private Long bkrId;
    public Long getBkrId() {
        return bkrId;
    }
    public void setBkrId(final Long bkrId) {
        this.bkrId = bkrId;
    }
    @Column(name = "KONTO_NR", nullable = false, length = 45)
    private String kontoNr;
    public String getKontoNr() {
        return kontoNr;
    }
    public void setKontoNr(final String kontoNr) {
        String oldValue = this.kontoNr;
        this.kontoNr = kontoNr;
        firePropertyChangeEvent("kontoNr", oldValue, this.kontoNr);
    }
}
```

(b)

```
import dbschema.xml // Das aus der Datenbank extrahierte
                    // Schema im XMI-Format

entity BuchungskreiseF
    (id=bkrId sequenceName=BKR_SEQ) {
        String kontoNr (notNull, length=45)
        Long rgNrBkrIdentifikator (notNull, length=1)
        String referenzcodeKontoNr (notNull, length=45)
    }
```

Abbildung 3: Generierter Programmcode für den Zugriff auf das DBMS aus Java mit der JPA (a) und das Datenmodell spezifiziert mittels der externen Datenmodell-DSL (b).

```
entity BuchungskreiseF
    (id=bkrId sequenceName=BKR_SEQ) {
}
```

Nur dann, wenn es Abweichungen gibt, beispielsweise zwischen der Namensgebung im Java-Programm und dem Datenbankschema, müssen diese explizit spezifiziert werden, um den Code aus Abbildung 3(a) zu generieren, wie es in Abbildung 3(b) zu sehen ist. Durch dieses Prinzip werden die DSL-Programme klein gehalten und die Ausnahmen sind explizit sichtbar. Abbildung 4 zeigt, wie die aus der Datenbank importierten Schemainformationen im Xtext-Editor genutzt werden können, um diese Schemainformationen in der Datenmodell-DSL bei Bedarf anpassen zu können.



Abbildung 4: Anpassung der Schemainformationen im Xtext-Editor.

4 Einsatz einer internen DSL für die Eingabeprüfung

Bei der Implementierung von Validierungsregeln für die Eingabeprüfung geht es im Wesentlichen darum Bedingungen zu definieren, deren Überprüfung geeignete Fehlermeldungen generiert, falls diese Bedingungen nicht erfüllt sind. Abbildung 5(a) zeigt die Eingabeprüfung mit Java, wie sie zunächst im vorgestellten Migrationsprojekt ohne eine speziell zugeschnittene interne DSL durchgeführt wurde. Auch hier gibt es für alle Eingabeprüfungen immer wiederkehrende Konstrukte. Nach eingehender Analyse der Überprüfungsfunktionen wurde eine interne DSL für Java entworfen, für die ein Beispiel in Abbildung 5(b) dargestellt ist.

Ein Kernkonzept zur Lösung dieses Problems besteht in der Möglichkeit, Boolesche Ausdrücke zu formulieren. Solche Ausdrücke können in Java selbst gut formuliert werden. Bei einer externen DSL müsste man dies neu implementieren, was sehr aufwändig ist. Daher macht es hier Sinn die Beschreibung von Validierungsregeln innerhalb von Java so prägnant wie möglich zu machen – sprich die API optimal in Form einer internen DSL auf das

(a)

```

addValidator(new Validator<Institutionen>() {
    @Override
    public ValidationResult validate(final Institutionen
        institution) {
        final ValidationResult result = new ValidationResult();
        if (institution != null && institution.getEsrNr() != null
            && !CheckUtils.checkPcKontoNrPruefziffer(
                Long.parseLong(institution.getEsrNr()))) {
            result.add(new SimpleValidationMessage(
                getResourceManager().getString("validation.esr.msg"),
                Severity.ERROR, getModel(Institutionen.DISC.
                    esrNr())));
        }
        return result;
    }
});

```

(b)

```

@Check
void checkEsrMsg() {
    if (!checkKontoNrPruefziffer(parseLong(this.getEsrNr())))
        error("validation.esr.msg", desc.esrNr());
}

```

Abbildung 5: Eingabeprüfung mit Java ohne interne DSL (a) und mit interner DSL (b).

Problem zu zu schneiden.

Dazu wurde zunächst eine Java-5-Annotation `@Check` eingeführt, mit der Validierungsmethoden ausgezeichnet werden können, siehe Abbildung 5(b). Die so annotierten Methoden werden dann durch ein speziell dazu implementiertes Framework reflektiv identifiziert und automatisch für Instanzen des angegebenen Typs (hier: `Konto`) aufgerufen. Bei genauerer Betrachtung der in Abbildung 5(a) gezeigten Validierungsregel fällt auf, dass viel Code durch geeignete Bibliotheksfunktionen vermieden werden kann. Beispielsweise kann das explizite Erzeugen von `ValidationResult` und `SimpleValidationMessage` implizit durch das Framework geschehen. Da die Methode vom Framework aufgerufen wird, reicht ein Aufruf einer `error(String, Object)`-Methode. Auch die Prüfungen auf Nullreferenzen, wie wir sie in Abbildung 5(a) finden, waren in allen bestehenden Validierungsregeln vorhanden. Die Semantik war dabei immer gleich: Wenn es eine Nullreferenz gibt, dann gilt die Bedingung nicht. Um auch diese Redundanz zu entfernen, wurde die Semantik der `@Check`-Methoden so geändert, dass `NullPointerException`s gefangen werden und dazu führen, dass der Check nicht ausgeführt wird. Über eine weitere optionale Annotation (`@NullableAware`) kann diese Semantik wieder abgeschaltet werden. Die in Abbildung 5(a) dargestellte Validierungsregel sieht in der internen DSL dann wie in Abbildung 5(b) dargestellt aus.

```

public class PersonenForm extends
Form<Personen> { ... }

public class PersonenHauptSubForm extends
SubForm<Personen> {

private JComponent vornameTextField;

@Override
protected void initComponents() {
    ...
    vornameTextField = builder.createTextField(desc.vorname(),
        Editable.PROPERTY_DEFAULT, MANDATORY);
    gepardBuilder.setNoLeadingBlanks (vornameTextField);

@Override
protected JComponent buildPanel() {
    TwoColumnsPanelBuilder builder =
        TwoColumnsPanelBuilder.instance(getBuilderFactory(),
            getResourceMap());
    ...
    builder.add("vorname", vornameTextField);
}
}

```

Abbildung 6: Programmcode für die Eingabemasken mit Java Swing (kleiner Ausschnitt).

5 Einsatz einer externen DSL für die Eingabemasken

Abbildung 6 zeigt einen kleinen Ausschnitt des Programmcodes für die Eingabemasken mit Java Swing. Dieser musste bisher manuell programmiert werden. Abbildung 7 zeigt unsere GUI-DSL, aus der nun der Java Swing Code aus Abbildung 6 generiert wird. Eine Beispielmaste der neuen *Gepard*-Anwendung, die mit Java Swing realisiert wurde, wird in Abbildung 8(a) dargestellt. Auch der Java-Swing-Code wiederholt sich vom Grundmuster her immer wieder, so dass es genau wie für die Datenzugriffsschicht aus Abschnitt 3 lohnenswert erscheint, diesen Code aus einer GUI-DSL zu generieren.

Für die Festlegung des konkreten Layouts einer grafischen Benutzungsoberfläche bietet es sich an, GUI-Editoren zu nutzen, die mit dem ergonomischen Konzept der direkten Manipulation arbeiten [SPC09]. Für Java Swing stehen bereits leistungsfähige GUI-Editoren zur Verfügung. Diese haben im hier beschriebenen Szenario allerdings einige Nachteile. Sie bieten immer alle Einstellungsmöglichkeiten der Java-Swing-Elemente an, und verstellen dadurch den Blick auf die wesentlichen Punkte, welche im betrachteten Projekt relevant sind. Die automatische Generierung von Formularen wird zwar durch Java Swing unterstützt, basiert aber dann auf Java Beans. Die Abstraktionen der Datenmodell-DSL wären damit nicht mehr nutzbar.

Im vorgestellten *Forms2Java*-Projekt wird aus der Datenmodell-DSL automatisch eine Standarddarstellung von Datenmodellen in Java-Swing-Formularen generiert. Die Dar-


```

model : gepard;
import "platform:/src/main/model/types.dao"
com.affichage.it21.gp.dao {
  flaechen {
    readOnly entity WaehrungF (id =(rvLowValue)) {
    }
    readOnly entity GepardVerwendungPvF (id =(pvOid)) {
      temporal manyToOne GeschpartnerAllBsF geschpartner ()
    }
  }
  verkauf {
    readOnly entity GepardVerwendungKdvtF (id =(kdvtId)) {
      temporal notNull manyToOne
        GeschpartnerAllBsF geschpartner ()
      notNull Number istLangfrist (castTo=Boolean)
      notNull Number istLokaldispo (castTo=Boolean)
    }
  }
}

```

Abbildung 7: Beispiel für die GUI-DSL.

stellung von Datenmodellen in Formularen sollte im *Forms2Java*-Projekt aber auch selbst kontrolliert werden können. Daher wurde eine eigene GUI-DSL entwickelt, in der die Elemente der Datenmodell-DSL referenziert und zusätzliche Layout-Informationen spezifiziert werden können, sofern sie von der selbst bestimmten Standarddarstellung abweichen. Um die Vorteile der direkten Manipulation zu nutzen, wurde für die GUI-DSL statt eines textuellen Xtext-Editors ein spezifischer graphischer Editor auf Basis des Graphical Modeling Frameworks¹¹ entwickelt, siehe Abbildung 8(b). Angemerkt sei, dass es hierbei um eine ergonomische Oberfläche für die Entwickler geht, die Ergonomie der damit erstellten Oberflächen für die Fachanwender (Abbildung 8(a)) ist ein weiteres wichtiges Thema, das wir an dieser Stelle nicht behandeln. In der Mitte von Abbildung 9 ist ein kleiner Beispielextrakt aus der GUI-DSL zu sehen, welcher die Datenmodell-DSL importiert. Die GUI-DSL stellt somit die Verbindung zwischen dem Datenmodell und der GUI her.

¹¹<http://www.eclipse.org/gmf/>

(a)

Schnellzugriff:
Geschäftspartner verw.
Favoriten verwalten

Aufgaben:
Zu Fav. hinzuf.

Person | **Erweitert** | **Finanzen** | **Verwendung**

☐ Verkaufspartner

Anrede, Briefanrede: Sehr geehrter Herr S.

Titel, Titel 2:

Nachname:

Vorname:

Strasse:

Strasse 2:

Postfach: ☐ Hat Postfach

PLZ, Ort:

Region, Land:

Gültigkeit:

Funktion:

☒ MWST - pflichtig

Bonität, Bem.:

Sprache:

Telefon:

Fax:

Mobile:

E-Mail:

Institutionen:

Haupt	Gründ-Jhr	Bezeichnung	Zusatz	Funktion	Gültig von	Gültig bis
<input type="checkbox"/>	515758	bernis AG			13.11.2008	

Institution:

☐ Haupt

Gültigkeit:

Vorschau:

Offene Dokumente:

(b)

Java - test/PersonenHauptform.gui_diagram - Eclipse Platform

Lucia Grande

PersonenHauptform.gui_diagram

☐ Verkaufspartner

Anrede, Briefanrede

Titel, Titel2

Nachname

Vorname

Strasse

Strasse 2

Postfach ☐ Hat Postfach

PLZ, Ort

Region, Land

Gültigkeit

Funktion

☐ MWST - pflichtig

Bonität, Bem.:

Sprache:

Telefon

Fax

Mobile

E-Mail

Palette

- Component
- Label
- Field
- ☒ CheckBox
- ☒ ComboBox

Properties

Component

Property	Value
Id	?

Outline

Abbildung 8: Beispielmaske (a) und GUI-Editor (b).

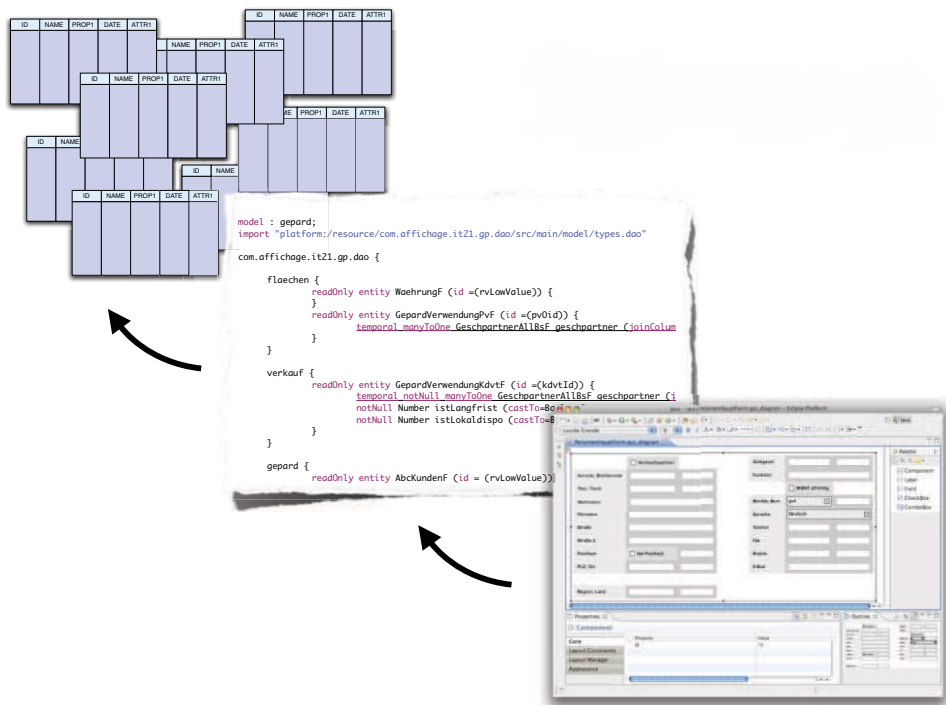


Abbildung 9: Die GUI-DSL als Verbindung zwischen dem Datenmodell und der GUI.

6 Wann lohnt sich der Einsatz welcher Art von DSL

Bei der Bewertung der modellgetriebenen Entwicklung im Vergleich zu traditionellem Vorgehen müssen grundsätzlich zahlreiche Faktoren beachtet werden. Da sich der Vorteil der modellgetriebenen Softwareentwicklung nur bedingt durch finanzwirtschaftliche Größen ausdrücken lässt, müssen sowohl monetäre als auch nicht monetäre Aspekte Berücksichtigung finden. Folgende, nicht monetär quantifizierbare Faktoren stellen einen Vorteil der modellgetriebenen Entwicklung dar [SVE07]:

1. Modellgetriebene Softwareentwicklung verschafft einen strategischen Geschäftsvorteil. Sobald eine DSL und die zugehörigen Transformationen und Generatoren für eine Plattform existieren, ist es möglich, sehr schnell neue Anwendungen aus demselben oder einem ähnlichen Problembereich zu implementieren.
2. Es wird primär fachliches Know-how zur Umsetzung neuer Anforderungen benötigt. Die Technologie tritt in den Hintergrund. Das Wissen über technische Projekt Rahmenbedingungen ist implizit im Generator abgelegt und kann sehr einfach wiederverwendet werden, ohne dass alle Mitarbeiter von vornherein ein tiefes Verständnis für die eingesetzte Technologie haben müssen.
3. Es findet eine klare und formale Trennung zwischen manuell erstelltem fachlichen und generiertem technischen Code statt (separation of concerns).
4. Automatisierung reduziert die Anzahl potenzieller Fehlerquellen.
5. Bestehende Modelle der Anwendung sind robust gegenüber Technologieänderungen. Dies entkoppelt den Anwendungslebenszyklus vom Technologielebenszyklus.

Darüber hinaus können verschiedene finanziell messbare Vorzüge erkannt werden, die teilweise eine Konsequenz der bereits aufgeführten Vorteile sind. Dazu zählen eine kürzere Implementierungsphase, geringere Kosten für den Einsatz neuer Technologien, reduzierte Kosten über den gesamten Produktlebenszyklus, geringerer Zeit- und Personalbedarf für die Umsetzung geänderter Anforderungen, eine kürzere Time-To-Market und geringere Wartungskosten. Die Menge des manuell erstellten Codes kann reduziert werden, woraus sich unmittelbar niedrigere Aufwände ergeben. Da der noch immer manuell in der DSL zu erstellende Code eine höhere Informationsdichte und Komplexität hat, als in traditionellen Projekten, kann nicht direkt von der Menge des manuell erstellten Quelltextes auf die Kosten geschlossen werden [SVE07]. Jedoch vergrößert sich das Einsparungspotenzial durch Wiederverwendung des Generators in mehreren Projekten (vgl. [RSB⁺04]). Eine Betrachtung der Aufwandsverteilung in klassisch durchgeführten Projekten in Gegenüberstellung mit dem modellgetriebenen Vorgehen zeigt, dass auf Grund der eingesetzten Werkzeuge ein Großteil der notwendigen, manuellen Tätigkeiten durch automatisierte Transformationen und Codegenerierung entfällt. Es ergeben sich zwar zunächst größere Aufwände zu Beginn der Projekte, doch wird dies durch die Ersparnisse im weiteren Projektverlauf aufgewogen.

Es stellt sich nun die Frage, für welchen Zweck sich der Einsatz welcher Art von DSL lohnt. Wir betrachten diese Frage für die drei Arten von DSLs, die im vorgestellten Projekt zum Einsatz kamen:

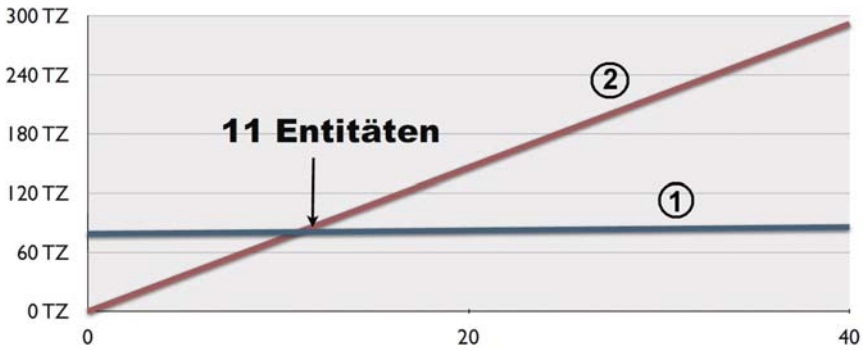


Abbildung 10: Vergleich der für das Datenmodell benötigten Programmzeilen mit DSL (1) und ohne DSL (2) in Abhängigkeit zu den implementierten Entitäten.

Externe DSL zur Beschreibung des Datenmodells Für die Entwicklung einer externen DSL ergeben sich zunächst initiale Kosten für die Implementierung dieser DSL, insbesondere auch für die Transformation in Programmcode. Dieser initiale Aufwand amortisiert sich erst bei einer bestimmten Größe des zu migrierenden (oder auch neu zu entwickelnden) Softwaresystems.

Für die Datenmodell-DSL aus Abschnitt 3 kann ein Vergleich des zu programmierenden Codes ohne bzw. mit DSL durchgeführt werden. Ohne DSL werden im Projekt Forms2Java durchschnittlich 7.000 Programmzeilen Java / JPA zur Programmierung des Zugriffs auf eine fachliche Entität benötigt. Für die Implementierung der DSL mit Xtext wurden 30.000 Zeilen Code programmiert, für die Implementierung der Transformation mit Xpand weitere 50.000 Zeilen Code. Für die Spezifikation des Datenmodells für eine Entität werden dann durchschnittlich jedoch nur noch 170 Zeilen je Entität benötigt. Rein rechnerisch, bei ausschließlicher Betrachtung der Programmzeilen, lohnt sich hier der Einsatz unserer Datenmodell-DSL ab 11 Entitäten, wie es auch in Abbildung 10 veranschaulicht wird. Die geringe Steigung der Linie 1 für die externe DSL ist in dieser Abbildung kaum zu erkennen. Durch das *Gepard*-System werden knapp 120 fachliche Entitäten verwaltet, die in gut 1700 Datenbanktabellen gespeichert werden.

Das Maß der Programmzeilen ist zum Vergleich des Aufwands natürlich nur sehr bedingt aussagekräftig, aber dieser Vergleich ist doch ein klares Indiz für die Wirtschaftlichkeit. Für die Betrachtung der initialen Kosten muss bei Bedarf auch der Aufwand für die Einarbeitung in neue Technologien (in unserem Projektkontext Xtext und Xpand) beachtet werden, was in obigem Vergleich nicht berücksichtigt wird.

Interne DSL für die Eingabeprüfung Für eine interne DSL muss keine Entwicklungsumgebung und keine Transformation implementiert werden, diese stehen z.B. für Java schon mit Standard-IDEs, JUnit und dem Java Compiler zur Verfügung. Damit ergeben sich initial geringere Kosten. Auch die Größe des Programmcodes kann deutlich sinken, wie der Vergleich von Abbildung 5(a) mit Abbildung 5(b) zeigt.

Ein weiterer Vorteil besteht darin, dass die Entwickler keine vollständig neue Spra-

che erlernen müssen, wie es bei einer externen DSL nötig ist (auch wenn sich externe DSLs sinnvollerweise an existierenden Sprachen orientieren werden). Dies kann die Akzeptanz bei den Entwicklern erhöhen.

Ein Nachteil interner DSLs besteht darin, dass es je nach Möglichkeiten der Wirtssprache Einschränkungen für die Mächtigkeit der DSL gibt. Java bietet hierzu nur recht beschränkte Möglichkeiten. Boo [Boo09] ist beispielsweise eine objektorientierte Programmiersprache, die besonderen Wert auf die Erweiterbarkeit der Sprache und ihres Compilers legt und somit bessere Möglichkeiten zur Entwicklung interner DSLs bietet als dies mit Java der Fall ist. Erwähnenswert sind bei Boo vor allem die sogenannten syntaktischen Makros, die die Sprache mit neuen Konstrukten erweitern können. Algorithmen können somit gekapselt und unter Benutzung eines neu eingeführten Schlüsselwortes wiederverwendet werden. Ein Problem ist bei Boo jedoch darin zu sehen, dass ein Softwareentwickler auf identischer Abstraktionsebene sowohl wiederverwendbare Code-Module spezifiziert, als auch als Sprachbestandteil zur Verfügung stellt.

Insgesamt empfehlen wir den Einsatz interner DSLs, wann immer dies ausreichend ist. Häufig wird jedoch wegen der beschränkten Möglichkeiten der eingesetzten Wirtssprachen der Einsatz externer DSLs erforderlich sein, um gute Produktivitätssteigerungen zu erreichen. Auch für eine interne DSL ist der Aufwand diese zu entwerfen nicht zu unterschätzen.

Externe DSL für die Programmierung der Eingabemasken Die obigen Aussagen zur Datenmodell-DSL treffen im Prinzip auch auf die GUI-DSL zu. Für die GUI-DSL wurde statt eines Xtext-Editors ein GUI-Editor entwickelt, siehe auch die Übersichtsdarstellung in Abbildung 2. Dieser GUI-Editor liefert keine neuen Möglichkeiten in der Programmierung. Vielmehr geht es hierbei darum, die Akzeptanz bei den Anwendungsprogrammierern durch eine angemessen zu bedienende Sicht auf die GUI-DSL zu erhöhen. Der Aufwand für die Konstruktion des GUI-Editors ist, auch mit dem Eclipse GMF, nicht zu unterschätzen; kann aber für die Akzeptanz bei den Mitarbeitern, und damit den Erfolg eines Migrationsprojektes, wie in diesem Papier beschrieben, von großer Bedeutung sein.

7 Verwandte Arbeiten

Die modellgetriebene Softwareentwicklung bildet die Grundlage für eine effiziente Verwendung von DSLs. Mit ihrer Hilfe können beispielsweise aus abstrakten Systemmodellen konkrete Systemartefakte generiert werden. Auf dieser Basis existieren verschiedene Vorgehensweisen, wie DSLs gestaltet und deren Tauglichkeit für einen konkreten Projektkontext evaluiert werden können. Die praktische Anwendbarkeit modellgetriebener Techniken und DSLs wurde hierbei in einer Reihe von Fallstudien empirisch untersucht. Für den speziellen Einsatzzweck der Migration von Softwaresystemen ist darüber hinaus der Aspekt der Transformation von Modellen von besonderer Bedeutung. Des weiteren existieren einige Vorgehensmodelle und Muster für die Migration, sowie auch für Anwendungsfälle, bei denen der Fokus auf der Evolution, Transformation und Migration von Datenbankan-

wendungen liegt. Dieser Abschnitt gibt einen Überblick über die vorgenannten Themenbereiche.

Das von uns präsentierte Migrationsprojekt basiert auf Methoden der modellgetriebenen Softwareentwicklung. Mehrere Arbeiten bieten hierzu eine Einführung, beispielsweise [BG05, Sch06, SVE07]. Hierbei wird die Anwendbarkeit an überschaubaren Systemen mit Artefakten, die durch sich häufig wiederholende Strukturen geprägt sind, verdeutlicht. Potenzielle Herausforderungen und Probleme im Hinblick auf die Entwicklung von komplexeren Softwaresystemen, wie etwa das im Rahmen des Projektes *Forms2Java* migrierte Altsystem *Gepard* der APG Affichage, untersucht [FR07]. Die Analyse wird hierbei von den Autoren entlang einer Kategorisierung der Herausforderungen in die drei Bereiche der (1) Modellierungssprachen, (2) Trennung der Belange in verschiedene modellrelevante Sichten und (3) Manipulation und Management von Modellbestandteilen, geführt.

Diverse Fallstudien evaluieren den Einsatz von modellgetriebenen Methoden und Techniken. Eine Laborstudie führten etwa die Autoren in [WSG05] durch, um das Potenzial von modellgetriebenen Methoden bei der Vereinfachung der Entwicklung von autonomen Systemen, die Enterprise Java Beans einsetzen, zu untersuchen. Darüber hinaus evaluieren mehrere Studien das Potential auch im industriellen Kontext [BLW05, Sta06].

Modellgetriebene Methoden kommen, wie auch im Rahmen des von uns beschriebenen Migrationsprojektes, häufig in Verbindung mit DSLs zum Einsatz. Eine grundsätzliche Unterscheidung erfolgt durch die Einordnung in interne und externe DSLs [Fow09], bei *Forms2Java* kamen beide Varianten zum Einsatz. Eine weitergehende Klassifizierung findet sich in [LJJ07]. Die Autoren bilden verschiedene DSL-Varianten anhand von gemeinsamen Merkmalen bezüglich des Einflusses auf die Sprache selbst, die potenziellen Transformationen, die Werkzeuge und die Entwicklungsprozesse. Muster zur Erstellung von DSLs werden in [Spi01] behandelt. Hierbei orientiert sich die Klassifizierung vornehmlich an dem Verhältnis einer DSL zu einer etwaigen Basis- bzw. Wirtsprache. Kriterien, die für die Erstellung von DSLs sprechen, untersucht [MHS05]. Die Autoren liefern darüber hinaus einen Literaturüberblick zu Kriterien, die für die Erstellung von Mustern relevant sind. Hierbei erfolgt eine Unterscheidung von Mustern nach Phasen im Entwicklungsprozess von DSLs. Die Entscheidungsphase liefert die Entscheidung für oder gegen die Erstellung einer eigenen DSL. In der Analysephase wird weitergehendes Wissen über die Problemdomäne erworben. Die Entwurfsphase beschäftigt sich mit dem Entwurf der DSL, die danach in der Implementierungsphase realisiert und in der Deploymentphase auf einer Zielpattform eingesetzt wird. Eine weitere Klassifikation von Ansätzen zur Entwicklung von DSLs findet sich in [Wil01].

Als weitere Grundlage für die modellgetriebene Migration von Softwaresystemen ist die Transformation von Modellen anzusehen, da ein Modell des Altsystems auf ein Modell des Zielsystems abgebildet werden muss. [SK03] klassifiziert Ansätze zur Modelltransformation hinsichtlich ihrer Eigenschaft, Modelle direkt manipulieren, exportieren und die Transformationsregeln explizieren zu können. Des Weiteren werden als wünschenswerte Merkmale von Sprachen zur Transformation von Modellen verschiedene Eigenschaften herausgearbeitet. Darunter beispielsweise die Fähigkeit, eventuell existierende Vorbedingungen, unter denen eine Transformation sinnvoll angewendet werden kann, zu beschreiben. Eine Taxonomie von Sprachen zur Modelltransformation beschreibt [MG06]. Sie soll unter anderem als Entscheidungsgrundlage zur Auswahl geeigneter Verfahren dienen.

Herausforderungen und Probleme bei der Evolution von Software fasst [vDVW07] zusammen. Für das Teilgebiet der Migration liefert [BLW⁺97] einen Überblick über gängige Vorgehensmodelle und Muster, wie das im Rahmen dieser Arbeit für den speziellen Projektkontext diskutierte Dublo-Migrationsmuster. Der fehleranfälligen Umstellung von einem Altsystem auf das komplett migrierte System in einem Schritt („Big-Bang“, oder auch „Cold Turkey“ Ansatz genannt) stellen die Autoren in [BS95] ihren Ansatz „Chicken Little“ entgegen, wobei eine inkrementelle Migration hervorgehoben wird. Ein Nachteil zeigt sich hierbei jedoch bei der Wartung und dem koordinierten Zugriff auf parallel gepflegte alte und neue Datenbestände. Diesen Aspekt adressiert die Methode „Butterfly“, die die zu der Koordinierung benötigten Gateways entfernt [WLB⁺97]. Ein weiterer Ansatz wird in [WSV05] beschrieben, der für die Migration unter anderem Techniken der dynamischen Programmanalyse einsetzt. Erfahrungsberichte zu dem Einsatz von modellgetriebenen Techniken im Kontext der Migration finden sich etwa in [FBB⁺07, ZG04].

Auch für die Evolution und Migration von Datenbanken existieren zahlreiche Veröffentlichungen. [BGD97] untersucht die Migration von relationalen Datenbankschemata auf objekt-orientierte Datenbanksysteme. Eine weitere Analyse der Evolution und Integration von Schemata findet sich bei [Cla94]. [MP03] legt einen Fokus auf umkehrbare Schema-Transformationsregeln. Den Einfluss von heterogenen Architekturen bei Datenbanken auf die Transformation von Schemata wird bei [MP06] analysiert. Das Mapping zwischen unterschiedlichen Schemata wird mittels Schema Matching durchgeführt. [DSDR07] präsentiert beispielsweise für diesen Zweck den Ansatz „QuickMig“. Eine Übersicht über bestehende Ansätze für das Schema Matching findet sich in [RB01].

8 Zusammenfassung und Ausblick

Anhand eines nicht-trivialen Migrationprojektes diskutierten wir den Einsatz von DSLs zur Migration von Datenbankanwendungen. Der primäre Vorteil einer externen DSL besteht in der Möglichkeit, eine optimale Abstraktion für eine Anwendungsdomäne bieten zu können. Der primäre Vorteil einer internen DSL besteht darin, dass keine vollständig neue Sprache entworfen und implementiert werden muss. Der primäre Vorteil des GUI-Editors ist in der Akzeptanz einer angemessenen Werkzeugunterstützung durch die Anwendungsprogrammierer zu sehen.

Der grundsätzliche Ansatz, mit domänenspezifischen Sprachen Modelle aus Altsystemen zu extrahieren, um diese zu modernisieren, wird beispielsweise auch in [IM09] vorgeschlagen. Eine Frage, die immer für derartige Projekte gestellt werden muss, besteht darin, ab wann sich der Einsatz von DSLs lohnt. Es stellt sich insbesondere die Frage, ob sich die Entwicklung von spezifischen DSLs schon für ein einzelnes Projekt lohnt, oder ob dies erst sinnvoll ist, wenn beispielsweise eine Produktlinie aufgebaut werden soll. Unsere Antwort dazu ist, dass es gar nicht immer sinnvoll ist zu versuchen, eine DSL zu entwickeln, die über viele Produkte hinweg genutzt werden kann. Der Grund dafür besteht darin, dass eine derartige DSL (üblicherweise) deutlich komplexer werden würde, als dies für ein einzelnes Projekt erforderlich wäre.

Um den Einsatz von DSLs schon für einzelne Projekte wirtschaftlich sinnvoll gestalten zu können, kommt dann der effektiven und effizienten Werkzeugunterstützung zur Implementierung der DSLs eine entscheidende Rolle zu. Mit den diversen Werkzeugen im Umfeld des Eclipse-Projektes stehen inzwischen leistungsfähige Werkzeuge zur Verfügung, um externe DSLs effizient implementieren zu können. Es ist prinzipiell mit diesen oder ähnlichen Werkzeugen möglich, eine DSL für mehrere Projekte zu nutzen, indem diese DSL für jedes Projekt individuell an dessen spezifische Bedürfnisse angepasst werden kann. Das Ziel sollte aus unserer Sicht dabei nicht sein, *eine* DSL für viele Projekte zu entwickeln, die schnell zu komplex werden könnte, sondern eine *Familie* von (kleinen) DSLs zu entwickeln.

Ein noch offenes Problem besteht dann darin, wie eine gute Wiederverwendung für die Implementierung der DSLs einer Familie ermöglicht werden kann. Dieser Frage widmen wir uns gegenwärtig im BMBF-geförderten Projekt Xbase.¹² Ziel des Projektes Xbase ist es, den anfänglichen Aufwand zur Implementierung der Infrastruktur für eine DSL erheblich zu reduzieren. Durch diese Maßnahme soll die modellbasierte Softwareentwicklung auch schon bei kleineren Projekten kosteneffizient eingesetzt werden können. Dabei werden immer wiederkehrende Aspekte der DSLs in Xbase allgemeingültig, anpassbar und einfach wiederverwendbar implementiert. Der Einsatz von Xbase verspricht hier weiteres Kosteneinsparungspotenzial, gegenüber der jetzigen Situation mit dem Einsatz modellgetriebener Entwicklungsmethoden.

Literaturverzeichnis

- [BG05] Sami Beydeda und Volker Gruhn. *Model-Driven Software Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [BGD97] Andreas Behm, Andreas Geppert und Klaus R. Dittrich. On the Migration of Relational Schemas and Data to Object-Oriented Database Systems. Bericht, University of Zurich, 1997.
- [BLW⁺97] J. Bisbal, D. Lawless, Bing Wu, J. Grimson, V. Wade, R. Richardson und D. O’Sullivan. An overview of legacy information system migration. Seiten 529–530, dec. 1997.
- [BLW05] Paul Baker, Shiou Loh und Frank Weil. Model-Driven Engineering in a Large Industrial Context - Motorola Case Study. In Lionel C. Briand und Clay Williams, Hrsg., *MoDELS*, Jgg. 3713 of *Lecture Notes in Computer Science*, Seiten 476–491. Springer, 2005.
- [Boo09] The programming language Boo. <http://boo.codehaus.org/>, 2009.
- [BS95] M.L. Brodie und M. Stonebraker. *Migrating Legacy Systems – Gateways, Interfaces and The Incremental Approach*. Morgan Kaufmann, San Francisco, CA, USA, 1995.
- [Cla94] Stewart M. Clamen. Schema evolution and integration. *Distributed and Parallel Databases*, 2:101–126, 1994.
- [DSDR07] Christian Drumm, Matthias Schmitt, Hong-Hai Do und Erhard Rahm. Quickmig: automatic schema matching for data migration projects. In *CIKM ’07: Proceedings of the*

¹²<https://kosse-sh.de/projekte/xbase/>

sixteenth ACM conference on Conference on information and knowledge management, Seiten 107–116, New York, NY, USA, 2007. ACM.

- [FBB⁺07] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas und Jean-Marc Jézéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt und Frank Weil, Hrsg., *MoDELS*, Jgg. 4735 of *Lecture Notes in Computer Science*, Seiten 482–497. Springer, 2007.
- [Fow09] M. Fowler. A Pedagogical Framework for Domain-Specific Languages. *Software, IEE-EE*, 26(4):13–14, jul. 2009.
- [FR07] Robert France und Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *FOSE '07: 2007 Future of Software Engineering*, Seiten 37–54, Washington, DC, USA, 2007. IEEE Computer Society.
- [HBG⁺08] Wilhelm Hasselbring, Achim Bündenbender, Stefan Grasmann, Stefan Krieghoff und Joachim Marz. Muster zur Migration betrieblicher Informationssysteme. In K. Herrmann und B. Brügge, Hrsg., *Tagungsband Software Engineering 2008*, Jgg. 121 of *Lecture Notes in Informatics*, Seiten 80–84, München, Februar 2008. Köllen Druck+Verlag.
- [HKRS08] Wilhelm Hasselbring, Stefan Krieghoff, Ralf Reussner und Niels Streekmann. Migration der Architektur von Altsystemen. In *Handbuch der Software-Architektur*, Seiten 213–222. dpunkt.verlag, 2. Auflage, 2008.
- [HRJ⁺04] Wilhelm Hasselbring, Ralf Reussner, Holger Jaekel, Jürgen Schlegelmilch, Thorsten Teschke und Stefan Krieghoff. The Dublo Architecture Pattern for Smooth Migration of Business Information Systems. In *Proc. 26th International Conference on Software Engineering (ICSE 2004)*, Seiten 117–126, Edinburgh, Scotland, UK, Mai 2004.
- [IM09] Javier Luis Canovas Izquierdo und Jesus Garcia Molina. A Domain Specific Language for Extracting Models in Software Modernization. In *ECMDA-FA 2009*, Jgg. 5562 of *LNCS*, Seiten 82–97. Springer-Verlag, 2009.
- [LJJ07] B. Langlois, C.E. Jitia und E. Jouenne. DSL classification. In *DSM '07: Proceedings of the Seventh OOPSLA Workshop on Domain-Specific Modeling*, Jgg. DSM '07, Seiten 28–38, 2007.
- [MG06] Tom Mens und Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).
- [MHS05] Marjan Mernik, Jan Heering und Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
- [MP03] P. McBrien und A. Poullovassilis. Data integration by bi-directional schema transformation rules. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, Seiten 227 – 238, mar. 2003.
- [MP06] P. McBrien und A. Poullovassilis. Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach. In Anne Pidduck, M. Ozsu, John Mylopoulos und Carson Woo, Hrsg., *Advanced Information Systems Engineering*, Jgg. 2348 of *Lecture Notes in Computer Science*, Seiten 484–499. Springer Berlin / Heidelberg, 2006.
- [RB01] Erhard Rahm und Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, 2001.
- [RSB⁺04] Dr. G. Rackl, Dr. U. Sommer, Dr. K. Beschorner, Heinz Koßler und Adam Bien. Komponentenbasierte Entwicklung auf Basis der Model Driven Architecture. *OBJEKTSpektrum*, (5):43–48, September / Oktober 2004.

- [Sch06] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer Society: Computer*, 39:25–31, 2006.
- [SK03] Shane Sendall und Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Softw.*, 20(5):42–45, 2003.
- [SPC09] Ben Shneiderman, Catherine Plaisant und Maxine Cohen. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson Higher Education, 5. Auflage, 2009.
- [Spi01] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.
- [Sta06] Mirosław Staron. Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. In Oscar Nierstrasz, Jon Whittle, David Harel und Gianna Reggio, Hrsg., *Model Driven Engineering Languages and Systems*, Jgg. 4199 of *Lecture Notes in Computer Science*, Seiten 57–72. Springer Berlin / Heidelberg, 2006.
- [SVE07] Thomas Stahl, Markus Völter und Sven Efftinge. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 2. Auflage, Mai 2007.
- [vDVW07] Arie van Deursen, Eelco Visser und Jos Warmer. Model-Driven Software Evolution: A Research Agenda. In D. Tamzalit, Hrsg., *CSMR Workshop on Model-Driven Software Evolution (MoDSE 2007)*, Seiten 41–49, Amsterdam, The Netherlands, March 2007.
- [Wil01] D.S. Wile. Supporting the DSL spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, 2001.
- [WLB⁺97] Bing Wu, D. Lawless, J. Bisbal, R. Richardson, J. Grimson, V. Wade und D. O'Sullivan. The Butterfly Methodology: a gateway-free approach for migrating legacy information systems. Seiten 200 –205, sep. 1997.
- [WSG05] Jules White, Douglas Schmidt und Aniruddha Gokhale. Simplifying Autonomic Enterprise Java Bean Applications Via Model-Driven Development: A Case Study. In Lionel Briand und Clay Williams, Hrsg., *Model Driven Engineering Languages and Systems*, Jgg. 3713 of *Lecture Notes in Computer Science*, Seiten 601–615. Springer Berlin / Heidelberg, 2005.
- [WSV05] Lei Wu, H. Sahraoui und P. Valtchev. Coping with legacy system migration complexity. In *Proceedings. 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005. ICECCS 2005.*, Seiten 600 – 609, jun. 2005.
- [ZG04] J. Zhang und J. Gray. Legacy System Evolution through Model-Driven Program Transformation. In *EDOC workshop on Model-Driven Evolution of Legacy Systems (MELS)*, Monterey, USA, September 2004.

Dissertationspreis

XML Query Processing in XTC

Christian Mathis*
christian.mathis@sap.com
SAP AG
Walldorf, Germany

Abstract: In the past, the development of a declarative, set-based interface to access data in a DBMS was a key factor for the success of database systems. For XML, the lingua franca for declarative data access is XQuery. This paper summarizes the XQuery processing concepts that have been developed in the XTC system (the XML Transaction Coordinator)—a native XML database management system. We step through all query processing stages: from parsing over query normalization, type checking, query simplification, query rewriting, and plan generation to the execution.

1 Introduction

The eXtensible Markup Language (XML) was designed as a technique for document representation and data exchange. With the success of this meta language, the volume of data represented in XML grew steadily, resulting in large document collections. Keeping such collections serialized as text in files or as BLOBs in relational database management systems is clearly a bad idea. The process of parsing the relatively verbose XML representation upon access is too expensive. Furthermore, loading large XML instances into main memory is often not viable and multi-user access with updates cannot be efficiently supported without dedicated access mechanisms to document substructures. Therefore, in the last decade, tailored XML database management systems have been developed that can compactly encode XML documents, that enable the transfer of substructures of a document into main memory, and provide for ACID transactions. The XML Transaction Coordinator (XTC) [HH07] developed at the University of Kaiserslautern is a prototype of such an XML database management system (XDBMS). XTC is a so-called *native* XDBMS, because all its internal structures are tailored to XML storage and processing, in contrast to systems that map XML to relational tables for storage and processing. In the past, the development of a declarative, set-based interface to access data stored in a DBMS (e. g., SQL for relational systems) was a key factor for the success of database systems in general. For XML, the lingua franca for declarative data access is XQuery.

This paper summarizes the XML query processing concepts in native XDBMSs that have been developed in the author's doctoral thesis [Mat09]. It highlights all stages of the query

*This work was conducted while the author was an employee at the Database and Information Systems Group (DBIS) at the University of Kaiserslautern.

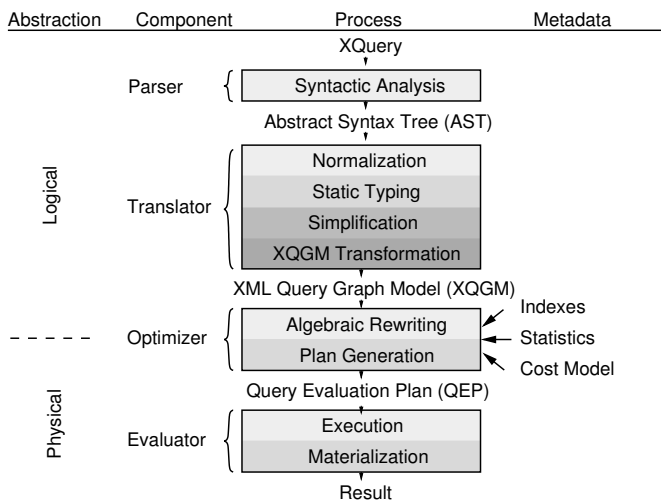


Figure 1: Query evaluation in XTC

evaluation process: from parsing over query normalization, type checking, query simplification, query rewriting, and plan generation to the final execution. This approach to query processing resembles the “standard” query processing pipeline of relational query processors and, in fact, this work borrows quite some concepts. However, the semantic richness of the XML data model and the XQuery language requires new solutions at most stages and poses many interesting research problems. By building on the “standard” pipeline and standard techniques, the work from [Mat09] can be integrated in existing relational query processors, for example, to enable XML management in relational engines.

2 XML Query Processing on XTC—An Overview

Given a declarative query, the query processor has to generate a semantically equivalent, cost-optimal, procedural program, which consists of algorithms and database-specific access methods. In the following, we will sketch the process of XML query processing in XTC, from the external representation of a query in the XQuery language to the execution on the data store.

In the late 1980s and in the 1990s, the DB research community spent substantial efforts on the development of *extensible* query processors for database systems. The idea was to provide for a framework into which new concepts, such as new language constructs, new data models, or new processing algorithms could easily be integrated without the need to re-implement large portions of a query processor [Mit95, KD99]. Systems like EXODUS [GD87], VOLCANO [GM93, Gra94], and Starburst [MKL88, HFLP89, PHH92] are some well-known examples from that time. The query processor developed in [Mat09] stands in the tradition of these systems. Therefore, many concepts and terms could be

borrowed, and, although the XTC query processor was built from scratch, it can be seen as an extension in the sense of the idea of extensible query processing. To cope with complexity, query processing is generally split up into a number of stages. Each stage receives a query representation generated by some preceding stage (or given as input) and produces a further representation with a lower level of abstraction but enriched with more specific information on how the query has to be evaluated. Figure 1 depicts all query evaluation stages of the XTC query processor.

The process has a *logical abstraction layer* and a *physical abstraction layer*. The logical layer is completely system independent. The query representations and actions at this level can be reused to implement a query processor for another XML data source. The aim at this layer is 1) to find a procedural internal representation such that semantically equivalent (but syntactically different) queries are mapped onto the same representation, and 2) to rewrite the query in a way such that intermediate results are minimized. Such a representation is a good starting point for the actions at the system-dependent physical abstraction layer below, because, in contrast to the declarative external query representation, a procedural internal representation contains more information about how the query can be evaluated. Furthermore, mapping semantically equivalent queries to the same internal representation makes the query processor robust.

At the physical layer, the query processor has to cope with low-level issues such as document storage layout, index structures, or processing algorithms to generate a program that operates on the database and efficiently computes the query result. In total, the query processor consists of the six components (see Figure 1): the *parser*, the *translator*, the *optimizer*, the *evaluator*, and the *metadata component* of the XTC system. Some of these components can share a sixth *infrastructure component*, which is not depicted in Figure 1. In the following, we give an overview over the various stages.

3 Parsing, Normalization, Static Typing, and Simplification

In the first stage, XQuery expressions need to be analyzed by a parser and to be converted into an *abstract syntax tree* (AST). In XTC, the XQuery grammar specified by the W3C Recommendation [BCF⁺04] is given to a parser generator to create the XQuery parser. In the next stage, the query translator transforms a given AST into an internal representation for the query optimizer. The translator has four stages: *normalization*, *static typing*, *simplification*, and *XQGM transformation*. Normalization and static typing are defined in the XQuery Formal Semantics Recommendation [CFS07]. Normalization transforms an XQuery expression to an equivalent expression in the XQuery Core Language, which is a subset of the original XQuery language. Static type checking derives the type of all subexpressions in the query and checks for static typing errors. The derived type annotations of all subexpressions can be used for optimization and restructuring.

Simplification aims at the removal of subexpressions with no effect on the query result. Such redundant constructs are sometimes introduced by programs that automatically generate queries, by view expansion, by users who do so accidentally, or by normalization.

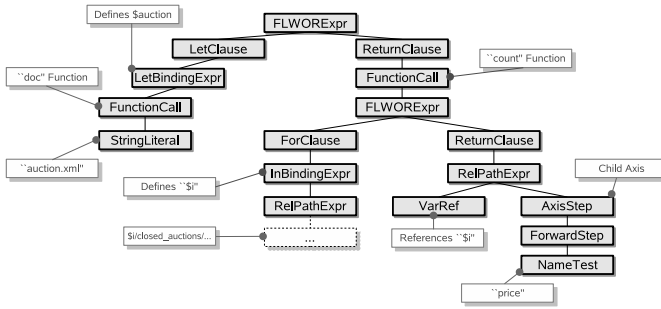


Figure 2: Abstract syntax tree for XMark query Q5

Simplification is implemented using the *infrastructure component* of the query processor. This component interprets a query representation (in this case the AST) as a tree and employs a rule-inference engine to apply tree transformations that are specified by *restructuring rules*. A rule has a *pattern* and a *transformation instruction*. When a rule matches the tree representation, the transformation instruction is applied to rewrite the tree at that position. Because the infrastructure component is just an implementation aspect, it will not be introduced in detail.

To illustrate these steps, let us consider the following query that emanates from the XMark benchmark [SWK⁺02] (Query 5) and returns the number of *price* elements that have a content larger than or equal to “40”:

```
let $auction := doc("auction.xml") return
count(
  for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)
```

The abstract syntax tree produced by the parser for this query consists of roughly 40 nodes. For the sake of brevity, Figure 2 does not contain all these nodes, but only a fragment of the complete AST. As you can see, the representation is quite straightforward. Every particle from the XQuery grammar corresponds to a node in the AST.

Normalization translates the AST produced by the parser into a rewritten AST with the same semantics, but with a reduced set of language constructs. As a result, normalization removes syntactic sugar. The normalized version of the above query has the following form¹:

```
let $auction := doc(auction.xml)
return count(
  for $i in ddo(
    for $fs:dot in $auction
    return ddo(
      for $fs:dot in child::site
      return
        ddo(
          for $fs:dot in child::closed_auctions
```

¹Note, this representation is simplified to facilitate comprehension. Function *ddo* stands for *fn:distinct-doc-order*, and—against the W3C recommendation—the constructs to produce positional information are omitted.


```

        return child::closed_auction)))
where fn:data(ddo(
  for $fs:dot in $i
  return
    ddo(for $fs:dot in child::price
      return child::text())) >= fn:data(40)
return
  ddo(for $fs:dot in $i
    return child::price))

```

You can observe that the normalized variant of the query does not contain any path expressions, only axis steps (e.g., `child::site`). Path expressions are rewritten to *for* clauses. The normalization process injects *ddo* and *fn:data* functions to ensure duplicate-free intermediate results (*ddo*) and atomic values for comparisons (*fn:data*).

Static typing infers the type of all subexpressions in a normalized query. For example, in the query above, the static type of the integer literal “40” is trivially *integer*. The surrounding *fn:data* function also delivers type *integer*, which is then used in the comparison. The comparison, in turn, is of type *Boolean*, and so on.

Even in our small example, you can observe that the normalization process is defined in a rather defensive manner, i.e., it injects certain functions blindly, even when they are not necessarily required. For example, the injected *fn:data* function around the integer literal “40” does not have an effect and can be safely omitted. A further example is the *ddo* function that is always injected, even when the intermediate result will always be in distinct document order. Besides normalization, users might write XQuery expressions with redundant or unnecessary subexpressions. Simplification aims at removing this kind of redundancy. An equivalent query for the above one might look like the following:

```

let $auction := doc(auction.xml)
return count(
  for $i in
    for $fs:dot in $auction
    return
      for $fs:dot in child::site
      return
        for $fs:dot in child::closed_auctions
        return child::closed_auction
where fn:data(
  for $fs:dot in $i
  return
    for $fs:dot in child::price
    return child::text()) >= 40
return
  for $fs:dot in $i
  return child::price)

```

The *ddo* functions are not necessary and the *fn:data* function around the integer literal can be removed². Currently, the XQuery processor can detect simplification opportunities in various situations (see [Mat09]). Note, however, that the simplification logic aiming at removing *ddo* functions is not yet integrated (although this topic has already been discussed in the literature [FHM⁺05]). Since XQuery is a quite flexible and freely composable language, many more situations than those handled in this work allowing for simplifications might exist. This work does however not dwell further.

²This is actually possible, because static typing revealed that the argument of the *fn:data* function is already an atomic value and therefore does not need atomization.

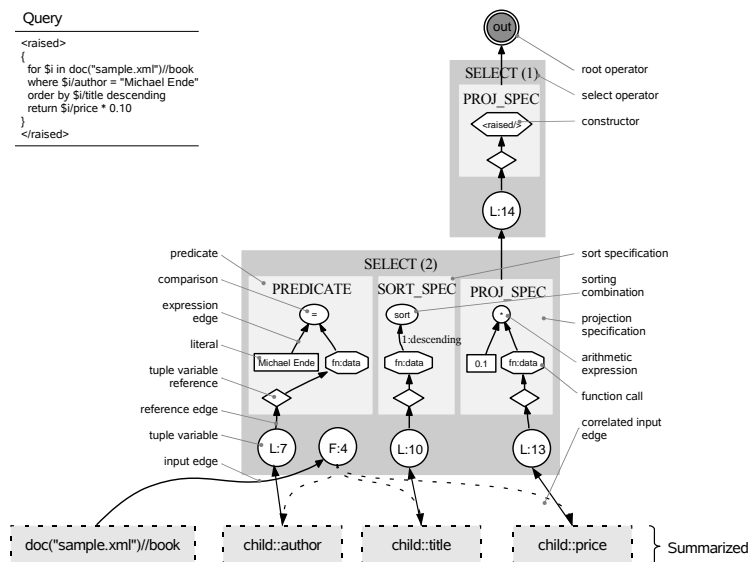


Figure 3: An example query represented in XQGM

4 The XML Query Graph Model

The last translation stage is the XQGM transformation. In this stage, the query translator transforms the AST into an instance of the XML query graph model. This is necessary, because the AST is not an appropriate format for query optimization as it lacks procedural-ity, i. e., it does not reveal data flow and control flow to evaluate the query. A better-suited internal query representation is the *query graph model* (QGM) introduced in the relational Starburst system [HFLP89, PHH92]. Although the QGM was designed for a relational engine, it provides enough flexibility to embed new language constructs like, for example, SQL recursion. In this work, we reused the QGM to support XML query processing. The resulting internal representation is called XQGM for *XML query graph model*. The initial XQGM instance for our sample query is depicted in Figure 4. All logical and physical plans presented in this and the following chapters are generated by a plan visualization tool developed in [MWHH08].

The syntax and semantics of XQGM can be found in [Mat09]. Here, we only give a brief introduction by example. Consider the query and its corresponding XQGM instance in Figure 3: An XQGM instance is an operator graph or a box-and-arrow diagram. Every box is a logical operator which produces data (most operators also consume data). The data produced flows along the arrows. All operators have a *name* describing the functionality of the operator and a unique *identifier* that follows the name in braces, e. g., “SELECT (2)”. In the following, we use a lower-case font to refer to operator names. The graphical elements inside an operator specify how the operator processes input data

and how it computes results. For example, *select* (2) consists of four so-called *tuple variables* (depicted as circles) controlling the input data flow and creating a tuple stream, a *predicate* describing the selection expression on the tuple stream, a *sort specification* to modify the order of the tuple stream, and a *projection specification* defining how the output shall be computed. Tuple variables carry a quantifier (e. g., “F”, for *for* quantification, and “L” for *let* quantification; see below) and a unique identifier to facilitate their distinction separated by a colon. The data model, based on which the semantics of XQGM is defined is similar to the XML data model (XDM) [FMM⁺04]. The major difference is that the XQGM data model allows tuples with nested tuple sequences.

To illustrate the semantics of XQGM, we step through the query execution of our sample query shown in Figure 4:

- Let us start with the control flow: The query processor calls the topmost *select* (1) operator, which, in turn, calls the next *select* (2) operator below to produce some output. *Select* (2) has three tuple variables, one of which carries an “F” specifying *for*-quantification semantics. The other tuple variables carry an “L” for *let*-quantification semantics. Tuple variables receive the output generated by their subgraphs. They define how this output is assembled into a stream of tuples. How this actually works will be sketched below. For now, we just proceed with the subexpression under tuple variable F:6. *Select* (3) is called and, in turn, *access* (5).
- Every operator calls its dependent sub-operators and awaits data for further processing. *Access* (5) is the first operator that actually produces data. It is a document access operator delivering the *virtual root node* [FMM⁺04] of the “auction.xml” document. This node is passed to the *select* (3) operator which binds it to tuple variable F:0 and calls *select* (6) to produce a result for tuple variable L:5.
- *Select* (6) in turn calls *access* (7), which is a navigational access operator. This type of access operator needs a *context node* as input from which the navigation starts. The context node is delivered by a *correlated input edge*, depicted as a dotted arrow. Tuple variable F:0 provides this input by passing the currently bound virtual root node to *access* (7). The result of the navigation on the child axis and the subsequent name test is a single *site* node. This node is passed to *select* (6) which binds it on tuple variable F:1 and calls *select* (8) to produce results for tuple variable L:4.
- *Select* (8) calls *access* (9) which delivers the *closed.auctions* element (exactly one in every XMark document) using the current node at tuple variable F:1 as correlated input. The *closed.auctions* element serves as correlated input for *access* (10) which returns all *closed.auction* elements below. These elements are passed to tuple variable L:3 which collects them all, puts them into a sequence, and binds this sequence as the current value (which is actually the semantics of the *let* quantification).
- The sequence is then passed to the projection specification, which applies the *ddo* function. A tuple variable may either be referenced via a correlated edge (dotted arrow) or by a so-called *tuple variable reference* depicted as a rhomb. The *ddo* function is also applied in *select* (6) and *select* (3) passing the sequence of *closed.auction* elements to tuple variable F:6.

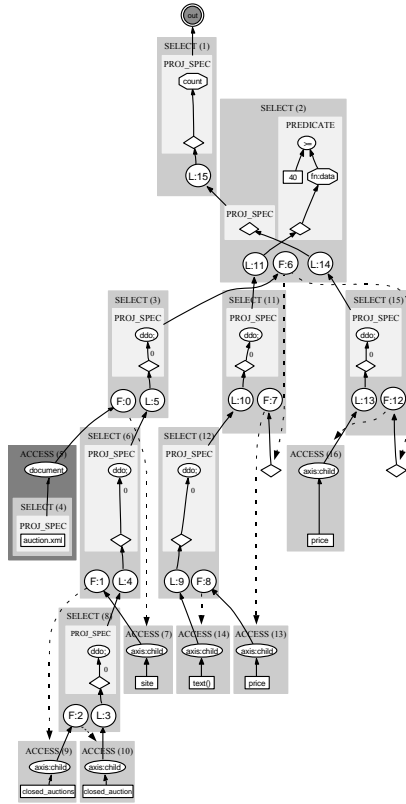


Figure 4: XMark query Q5 represented in XQGM

- So far, every *for*-quantified tuple variable received only a single node as input. For single nodes, the semantics of *for* and *let* are the same. This time, however, tuple variable F:6 receives a sequence of possibly more than one node. While *let* passes these nodes as a whole as described above, *for* iterates over the sequence items, just like the corresponding constructs in the XQuery language. You can further notice that the subtrees below tuple variables L:11 and L:14 depend on the current node at tuple variable F:6, because these subgraphs have a correlated input edge starting at F:6. This means that for every node at F:6, the dependent subtrees are evaluated and their result sequences are bound to the corresponding tuple variables. In the following, we will call L:11 and L:14 *dependent tuple variables*, whereas F:6 is called *independent*. The subtrees below independent tuple variables have to be evaluated first, because they provide the input for the subtrees below dependent tuple variables. Essentially, the subtree below F:6 evaluates `doc("auction.xml")/site/open-auctions/open-auction`. For every *open-auction*, the expression below L:11 evaluates the relative path `price/text()` and L:14 the relative path `price`.

- Inside `select (3)`, the predicate is evaluated for every *open_auction* element. If the predicate evaluates to *true*, the current value at tuple variable `L:14` is read by the projection specification and passed as an intermediate result to `select (1)`. In turn, `select (1)` collects all these intermediate sequences in another sequence on which the *count* function is evaluated to obtain the final result.

The reader familiar with the dynamic evaluation phase specified in the Formal Semantics has noticed that the evaluation model defined there and in XQGM is essentially the same, i. e., an initial XQGM instance acts as specified in the Formal Semantics. This is meaningful, because it ensures correctness. In a way, XQGM is a graphical representation for normalized XQuery expressions. A large fraction of XQuery can be captured solely by XQGM's select and access operators. We will not formally introduce the syntax and semantics of XQGM in this paper. The details can be found in [Mat09].

5 Algebraic Rewriting

Because the semantics of an initial XQGM instance generated by the query translator adheres to Formal Semantics Recommendation, the above sketched evaluation model heavily relies on nested subexpressions and *node-at-a-time* navigational methods. This model it is often far from being optimal.

Therefore, besides classical algebraic optimizations such as selection push-down and select fusion to minimize intermediate results and the number of operators required, the algebraic rewriting stage tries to unnest queries as far as possible to enable bulk or set-at-a-time processing. Unnesting substitutes correlated subexpressions by joins, i. e., by bulk operators. Like simplification, algebraic rewriting is also implemented using the infrastructure component. The XQGM instance is interpreted as a tree structure on which the generic rule engine executes rule-based transformations. The result of the algebraic rewriting stage is an unnested and pre-optimized XQGM instance. At this point, the physical optimization of the query begins and system-specific issues come into play. Before we discuss plan generation, we like to summarize the algebraic rewriting rules developed for the XTC query processor. [Mat09] contains all rules with the description of the rule pattern, its preconditions, and the transformation instructions:

- *Removal of external tuple variables*: Every variable reference in XQuery results in a tuple variable reference in XQGM. Some of these references are unnecessary and are removed by this rule.
- *Removal of descendant-or-self steps*: Due to normalization, a double-slash operation as in `doc("auction.xml")//item` in XQuery results in a *descendant-or-self::node()* navigation in XQGM. Sometimes, this navigation step can be replaced by a *descendant* step, which is achieved by this rule.
- *Range-query detection*: Many XDBMSs provide index structures to evaluate content-based predicates. Those predicates can be point queries or range queries. Range-based predicates are specified in XQuery with the help of comparison operators and

Example	Twig	Document	Result	Tuple Result
a) plain Plain twig nodes, order in document is not significant, all nodes produce output.				$[a_1, b_1, c_1, d_1]$ $[a_1, b_2, c_2, d_2]$
b) output ordering Plain twig nodes, order in document is not significant, all nodes produce output.				$[a_1, b_1, c_1]$ $[a_1, b_1, c_2]$ $[a_1, b_1, c_3]$ $[a_2, b_2, c_1]$ $[a_2, b_2, c_2]$
c) boolean Plain twig nodes and and/or twig node, all nodes produce output.				$[a_1, b_1, c_1, d_1]$ $[a_1, b_2, c_2, ()]$ $[a_1, b_3, (), d_2]$
d) optional Plain twig nodes and optional edge (dashed); subtwig rooted at 'b' is optional.				$[a_1, b_1, c_1, d_1]$ $[a_1, (), (), ()]$ $[a_1, (), (), ()]$
e) grouping Plain twig nodes and grouping node (double circle); subtwigs rooted at 'c' and 'd' grouped into result of 'b'.				$[a_1, b_1, <c_1, c_2>, <d_1, d_2>]$
f) output expression Plain twig nodes and grouping node (double circle); optional edge and output expression; only 'b' produces output.				$[<x>\{d_1\}\{d_2\}</x>]$
g) output filter Plain twig nodes and grouping node (double circle); optional edge and output filter (predicate).				$[d_1]$
h) positional Plain twig nodes and one output node; two nodes generating positional information and two positional predicates.				$[b_1]$

Figure 5: Twig matching examples

the Boolean *and*. The range-query detection rule finds such range predicates and converts them into an XQGM range predicate, which is easier to evaluate and to map to the above mentioned index structures.

- *Select fusion*: Some rewriting rules leave select operations in a state, where only a simple operation, like applying the *distinct-doc-order* function, is executed. In these cases, the select fusion rule merges the select operation with its input operation.
- *Predicate push-down*: Predicate push-down is a standard rewriting strategy from relational query engines. It can also be implemented in XQGM for XQuery. Due to the existential semantics of general XQuery comparisons (see [CFS07]), predicate push-down is a little bit more complicated to implement.
- *Query unnesting*: The normalization phase introduces a nested sub-expression, whenever a variable is referenced. This is also reflected in the XQGM instance of the query. Especially for navigation axes, this approach leads to node-at-a-time evaluation, i. e., for every input node, the navigation axis is evaluated as a sub-expression. A similar situation arises in SQL queries with nested sub-queries. In almost all situations, these queries are unnested by the SQL processor and are replaced by a join-based equivalent. This approach is also viable in XQGM. Here, however, we do not introduce value-based joins, but *structural joins*. For structural joins, many efficient implementations have been proposed in the literature (e. g., [AkPJ⁺02, CVZ⁺02, MHH06, MH06]). After the query has been unnested, all these algorithms can be applied. Besides from the discussion of query unnesting in [Mat09], an algebraic approach to query unnesting can also be found in [Mat07].
- *Twig detection*: Algorithms for twig pattern matching have been heavily researched in the past [BKS02, CLL05, FJSY05, CLT⁺06]. Twig matching algorithms can be used to evaluate branching path expressions that often occur in XQuery. To support twigs, XQGM specifies a dedicated twig operator. This operator has a so-called twig specification that can express twigs with various interesting properties that support idioms frequently occurring in XQuery expressions. Figure 5 exemplifies the semantics of the twig specification. The result is represented as graphical subtrees and as nested tuples (a data type of the XML algebra [Mat09] based on which XQGM is defined). Essentially a twig can return all nodes that match (Figure 5a). We can enforce that the result adheres to the document order [CFS07] (Figure 5b). The twig specification can define Boolean predicates (Figure 5c) and optional sub-patterns (Figure 5d). Some queries implicitly group results. Therefore, the double circle in Figure 5e signals that the matches below shall be grouped (in the tuple result, groups are represented by sequences in angle brackets). Furthermore, the XQGM twig specification allows to embed output expressions and filter predicates, for example, to generate new XML elements based on the matched results (Figure 5f) or to check content-based predicates (Figure 5g). Finally, even positional predicates can be specified (Figure 5h). The twig detection rule is responsible to find sub-structures in an XQGM instance that can be evaluated by a twig operation with the expressiveness of the sketched twig specification.

To illustrate the rewriting stage, Figure 6 shows the results on our running example. First you can observe that the query does not contain any nested subexpressions, it has been completely unnested (note, the dotted lines inside `twig (28)` have a different semantics).

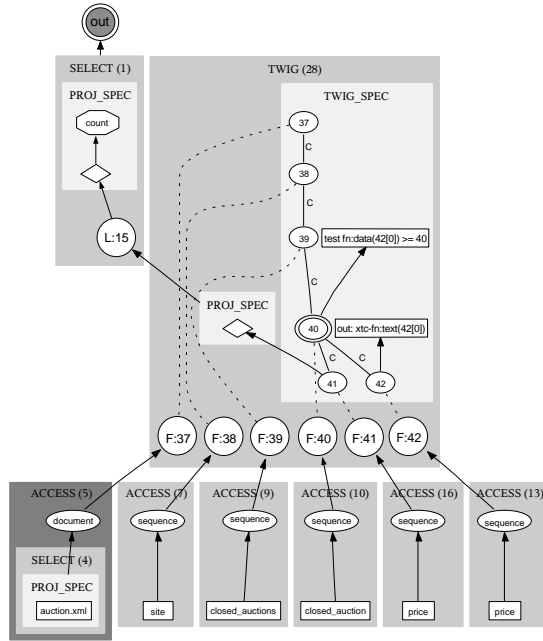


Figure 6: Rewritten XMark query Q5

The access operators are not navigation-based anymore, but access *all* nodes that match a certain node test (e. g., all *price* nodes). The nodes in the twig specification are connected to the corresponding input tuple variables by dotted lines. In the specification, *C* stands for a child relationship, *D* for descendant, and *@* for attribute. The *select (1)* operator has the same function as before. It collects the *price* sequences generated by the twig operator, adds them to a sequence, and applies the *count* function. Note, the completeness of the unnesting and twig detection rule has not been formally shown in [Mat09], i. e., we do not know, whether all twig queries can be unnested and whether all twigs can be found. Therefore, we classify the approach as best effort. Nevertheless, we note that all XMark queries [SWK⁺02] could be unnested and all twigs were discovered.

6 Plan Generation

Given the result of the rewriting stage, the query processor now has to assemble a query execution plan (QEP), i. e., it has to map the logical operators onto algorithms and document access methods. These algorithms can roughly be grouped into 1. navigational, join-based, and index-based methods for path matching, and 2. into all remaining algorithms that are necessary to evaluate selections, projections, grouping, value-based joins,

etc. The algorithms of the first group, which are also called *path processing operators* (PPOs), play a major role in this work, because PPOs access the document (in contrast to the operators in the second group, which merely operate on the intermediate results delivered by path operators). Document access can be expensive, therefore, these operators need special attention. The set of all physical operators is called *physical algebra* (PAL). This term was introduced in [GM93] and shall help to distinguish operators from the physical level (algorithms) from operators on the logical level (XQGM). We give a brief introduction to the physical algebra in the next section.

Given an XQGM instance, plan generation is implemented in two stages, the first one of which also relies on the rule engine of the infrastructure component. Here, the rules describe logical-to-physical mappings or XQGM-to-Plan transformations (similar to [KD99]). Whenever a rule matches, a description of the physical operators implementing the matched XQGM operator is created and attached to the matched logical operator. Considering the relationship between a logical XQGM operator and operators from the physical algebra, the 1:1, 1: n , n :1, and n : m cardinalities apply: Sometimes there is only one physical alternative for a logical operator (1:1), sometimes there are more than one alternatives (1: n), and sometimes a group of logical operators is implemented by a (group of) physical operator(s) (n :1 or n : m). In the second stage, the plan generator iterates over the XQGM instance and builds different QEPs from the physical alternatives it finds. Often, the optimizer can create a large variety of structurally different but logically equivalent QEPs for a single XQGM instance.

From all the different QEPs, the query processor now has to decide, which of them is the cheapest in terms of processing costs. The answer to this question depends on a large variety of parameters, such as the optimization goal (e.g., response time, throughput, main-memory usage, etc.), the structural layout of the document, value distributions in the document, the current system state (I/O-bound or CPU-bound), and so on. The applicability of certain physical operators depends on the physical layout of the database, i.e., on document storage and indexing. At the time [Mat09] was written, cost-based query optimization was under development. Therefore, in [Mat09], the author restricted the plan generator to the following: 1. The plan generator should be able to generate every possible plan in the search space, and 2. the plan generator be able to find a good plan based on simple heuristics. We will come back to this point at the end of the next section.

7 The Physical Algebra

The physical algebra contains all query evaluation algorithms. Of particular importance are those algorithms that access the document or some index to match path patterns. Because XQuery heavily depends on efficient path pattern matching, we focus our discussion on path procession operators (PPOs). We distinguish *navigational*, *join-based*, and *index-based* PPOs. The first group of operators is also the most expressive one; every path expression in a query can be evaluated by navigating the document. Compared to join-based and index-based methods, they are, however, often enough the group of operators with the slowest performance. Hence, navigational primitives are a “fall-back solution”, when no operators of the other two groups can be applied to evaluate a certain path expression.

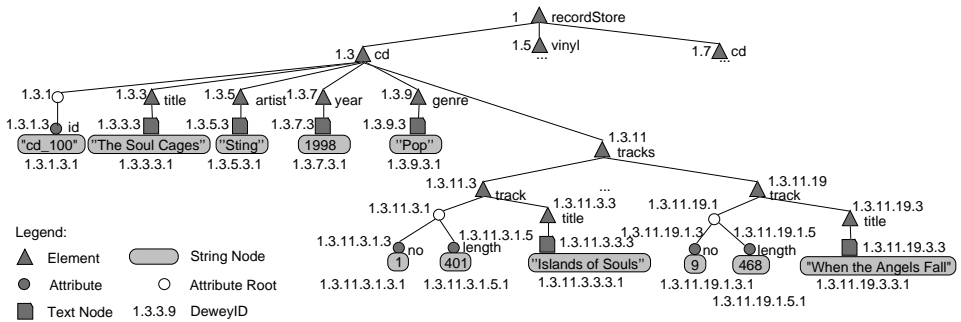


Figure 7: Sample document with path-based node labels

Join-based operators stream through the document or over an index that contains references to all elements and evaluate path expressions by matching structural relationships among the streamed nodes. Compared to navigational methods, they often provide for better performance. However, their use is restricted to certain XPath axes. The two most prominent representatives for this group are *structural joins* (STJ) and *holistic twig joins* (HTJ). Especially holistic twig joins have gained much attention in the literature and many variations of the original algorithm [BKS02] have been presented. Most of these variations aimed at optimizing the algorithm’s structure matching phase and at increasing its performance. Below, we will sketch an HTJ algorithm that has been designed to work hand in hand with indexes and the other algebra operators.

The last group of operators provides index access. We will illustrate how path queries extracting inner elements (such as `//cd[id="cd_101"]`) can be answered with path indexes and how index-based operators can be “married” with join-based operators. Note, index-based operators have yet again a reduced expressiveness compared to join-based operators, because join-based operators can match arbitrary branching path patterns and index-based operators can only match linear paths. In the following, we will give an overview over the PPOs in the physical algebra, starting with navigational PPOs.

7.1 Navigational PPOs

Let us assume, a document like the one depicted in Figure 7 is stored in XTC’s document store [Mat09]. Upon storage, the nodes are numbered using path-based node labels (also known as DeweyIDs or OrdPaths). These labels have certain salient features: they allow to compute all ancestor labels, their lexicographical order represents the document order, and they leave gaps to insert new nodes without altering existing labels. Given a node label, the document allows to retrieve other nodes, for example, all children, the complete subtree, the next/previous child, the parent node, and so on. These access primitives are used by navigational PPOs.

The *axis-step navigation operator* receives an axis, an XQuery node test, and a reference to the operation that provides its input node stream. For each node in the input, the algorithm queries the document store to evaluate the axis expression and applies the node test. Nodes that fulfill the test are passed on to the next operator. With this algorithm, all XPath axes and node tests can be evaluated. Note, this algorithm can be applied to evaluate all (but the left-most) *access* operations in Figure 4.

A problem with the axis-step navigation operator is that it might do work twice and that it might return duplicates in the wrong order (depending on the query and the document). As an example, consider XQuery expression $\$d/\text{descendant}::w$, where $\$d$ is bound to sequence S . S is a series of context nodes that serve as starting points for the navigation. Let S contain two nodes u and v , where v is a descendant of u . Suppose that in the document, a node w exists, which is a descendant of both u and v . On the evaluation of axis step $\$d/\text{descendant}::w$, the above algorithms would return w twice (because they are evaluated both on u and v). However, XQuery semantics demands that the result of an axis step is duplicate free. Therefore, *distinct-doc-order* functions are embedded during normalization to sort the result and remove duplicates. The *multi-node navigation operator* avoids sorting and duplicates by analyzing the input sequence and only navigating from nodes that produce a duplicate-free result in document order. The details of this and the following algorithms are omitted in this paper. We refer to [Mat09].

7.2 Join-Based PPOs

Navigational PPOs require some input node(s), which serve(s) as a starting point for the navigation operation over the document. In contrast, join-based PPOs do not directly access the document. They operate on *two or more* node streams and are capable of finding path matches in these streams. How the node streams are created does not matter. They could be the result of a document scan, an index scan (see below), or they could be the result of other operators. XTC implements variations of two well-known algorithms: the structural join (STJ) and the holistic twig join (HTJ) algorithm.

The structural join in XTC is a merge-join algorithm, which is an extension of the original StackTree operator [AkPJ⁺02]. As an example, consider the two tuple streams containing *track* elements and *title* elements (see Figure 7). The *track* stream has the following node labels: 1.3.11.3, 1.3.11.5, 1.3.11.7, and so on. The *title* stream has labels 1.3.3, 1.3.11.3.3, 1.3.11.5.3, and so on. Suppose we want to find all *title* children below all *track* elements. Because our node labels encode this information and the node streams are ordered, we can apply the merge-based StackTree algorithm. In our example, the first *title* element does not find a join partner.

We extended the original StackTree operator by some features for the integration into our physical algebra. Our StackTree variant supports semi-joins, full joins, and outer joins. The join implementation returns the result in distinct document order (in case of a semi-join), or in *inner/outer* and *outer/inner* output ordering (in case of a full join or an outer join). The evaluation axis can be one of the following: *child*, *attribute*, *descendant*, *parent*, *ancestor* (and the *-or-self* variations of *descendant* and *ancestor*). The reverse variants of the algorithms are implemented by exploiting join commutativity.

The twig join operator is a complex merge-join operation that can be seen as an extension of the structural join. In contrast to structural joins, the algorithm can consume more than two node streams, in which it matches complex branching path patterns known as twigs. Our notion of a twig has been introduced as the twig specification in Section 5 and Figure 5. To the best of our knowledge, [Mat09] published the first algorithm that provides all features that can be expressed in our twig specification. Such an algorithm is desirable, because the higher its expressiveness, the more operations can be embedded into the twig algorithm, thus, resulting in a smaller number of operators in the final plan and fewer intermediate results. Furthermore, evaluating as many operations as possible can avoid intermediate-result materialization.

We picked a promising approach as the basis for our implementation, namely, the TwigOpt algorithm proposed by [FJSY05]. The algorithm operates on a set of node streams, one for each node in the twig. For example, in the physical representation of the XQGM instance shown in Figure 6, each twig input produces such a stream. The streams can be generated by a document scan, an index scan, or by other operators. They have to return the nodes in document order. The twig algorithm accesses a stream through a cursor. The cursor state can be modified using methods `setToFirst()`, `getCurrent()`, and `forwardTo()`. The first method initializes the cursor to the first node. The second method returns the node at the current location of the cursor, and the `forwardTo()` method advances the cursor. Based on the state of the cursors, the TwigOpt algorithm identifies the cursor that can skip the largest fraction of its input stream. This cursor is advanced as far as possible. After each move, the cursor positions are checked for a twig match. In case of a match, some output according to the twig specification is produced.

7.3 Index-Based PPOs

XTC supports a variety of indexes. The document store itself is an index that allows to retrieve nodes by their labels. All elements with a certain name can be indexed by the so-called *element index*. Element-index scans can produce node streams for the STJ and HTJ algorithms. For value-based point and range predicates, XTC can index text nodes in the *content index*. Because path expression occur frequently in XQuery, *path indexes* can be created. A path index is specified by a linear (i.e., non-branching) path pattern, for example `I(/cd/title)`. All nodes that fulfill this pattern are stored in the index. Combining path indexes with content indexes results in the so-called *content-and-structure* (CAS) index, which is also defined by a linear path pattern. For all these indexes, appropriate access operators exist in the physical algebra.

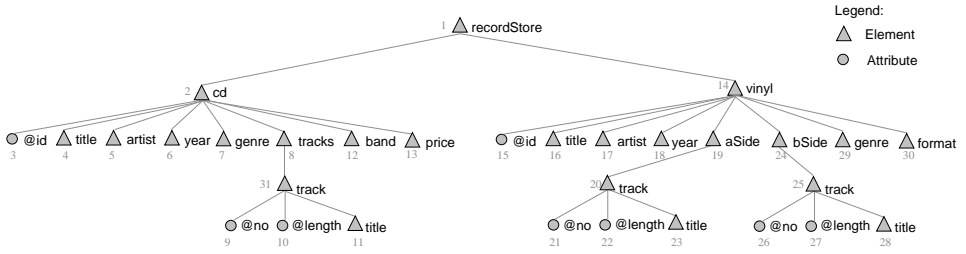


Figure 8: The path synopsis

In the following, we want to highlight a specialty of XTC, namely, the integration of path-index and CAS-index scans with the holistic twig join operator. Both index types are built with the help of the *path synopsis* (PS). A PS is the structural summary of all paths in the document. In XTC, the PS is kept in main memory for fast access. Figure 8 shows the PS of our sample document from Figure 7. Every node or attribute in the PS is labeled with a unique integer called *path-class reference* (PCR). Given the PS, a PCR, and a node label, we can reconstruct the entire path to the root without accessing the document (the labels can be computed from the given node label and the element names can be retrieved from the path synopsis). For example, PCR 11 (see Figure 8) and node label 1.3.11.3.3 (see Figure 7) let us reconstruct the following ancestor nodes: *title* (1.3.11.3.3), *track* (1.3.11.3), *tracks* (1.3.11), *cd* (1.3), and *recordStore* (1).

Because our path-based indexes store PCRs together with node labels, scan operations on these index types return PCR-label pairs. By applying ancestor reconstruction to such an index scan, we can compute the node streams that are required as input to the holistic twig join operator without document access. This is accomplished with the help of an algorithm called *ancestor tuple builder* (ATB). Figure 9 gives a schematic overview over the interaction between the holistic twig join and the ancestor tuple builder: Let us assume that a linear path pattern of a twig specification is covered by a path index (the darker shaded nodes in Figure 9). Their cursors (C_1 to C_3) forward the HTJ's cursor requests to the ATB-input algorithm, which returns the necessary nodes based on an index-scan cursor (C_I). Nodes for the ancestor cursors are computed as illustrated above. To trigger their computation, an ATB cursor can call `open()` to open the node stream and `processTo()`, which advances the computation to a given node label (similar to the other twig cursors).

7.4 Heuristics for Plan Generation

As stated in Section 6, the plan generator shall enable the generation of all plans in the search space (see [Mat09] for the details), and shall be able to assemble a good plan. To reach the second goal, we conducted a number of experiments based on the XMark benchmark [SWK⁺02]. Figure 10 highlights some results. All 20 XMark queries were evaluated either fully implemented by navigational operators (see Section 7.1), by join-based operators (see Section 7.2), or based on a set of existing indexes (see Section 7.3). As

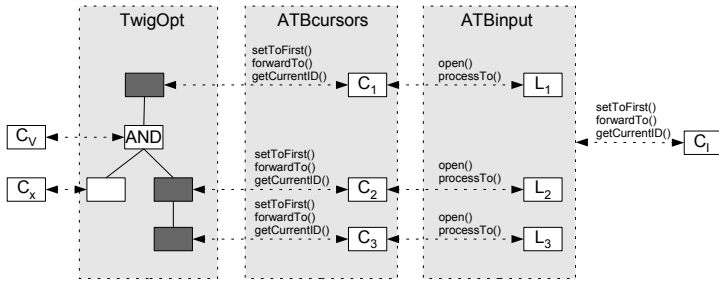


Figure 9: Integrating index scans with the twig join algorithm

you can observe, in this benchmark, join-based query evaluation is almost always a better strategy than navigation. Especially on queries with long paths, index-based evaluation is advantageous. From these and from other experimental results reported in numerous papers about indexing, structural joins, and holistic twig joins, we drew a set of rules to parameterize the plan generator. We chose the following simple heuristics: The plan generator 1. always unnests the query to enable join-based query processing, 2. favors join-based processing over navigational processing, if an element index exists, 3. favors twig joins over structural joins, and 4. gives indexes the following precedence (from high to low priority): CAS index, content index, path index, element index, and document index.

8 Conclusion

XML data processing has been an actively researched topic in the last decade, which lead to XML support in all major commercial database systems. This paper summarized the author's research on XML query processing, which was conducted during the authors engagement in the XTC project at the University of Kaiserslautern. The hierarchical data model and the semantically rich XQuery language required new approaches to data storage, indexing, query processing algorithms, and query rewriting. To bring these new ap-

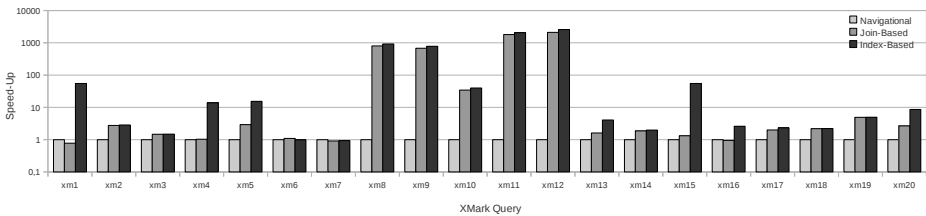


Figure 10: A comparison between physical operators on the XMark query set

proaches together, XTC leans on the classical relational query-processing pipeline and extends the well-known relational Query Graph Model for query representation. For a prototypical system, XTC has a quite extensive physical algebra including a rich set of different index types and navigational, join-based, and index-based query processing algorithms. This makes XTC ideal as a test bed for future research.

Acknowledgements

This work would not have been possible without the help of the XTC team: I like to thank Michael Haustein (the founder), Karsten Schmidt, Sebastain Bächle, Yi Ou, Leonardo Ribeiro, Aguiar Moraes Filho, Andreas Weiner, Stefan Hühner, and Caesar Ralf Franz Hoppen. I thank Theo Härder for his guidance and inspiration.

References

- [AkPJ⁺02] Shurug Al-khalifa, Jignesh M. Patel, H. V. Jagadish, Divesh Srivastava, Nick Koudas, and Yuqing Wu. Structural joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE*, pages 141–152, 2002.
- [BCF⁺04] Scott Boag, Donald Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation, 2004. <http://www.w3.org/TR/xquery/>.
- [BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD*, pages 310–321, 2002.
- [CFS07] B. Choi, M. Fernández, and J. Siméon. The XQuery Formal Semantics: A Foundation for Implementation and Optimization. W3C Recommendation, January 2007. <http://www.w3.org/TR/xquery-semantics/>.
- [CLL05] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In *Proc. SIGMOD*, pages 455–466, 2005.
- [CLT⁺06] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig2Stack: Bottom-Up Processing of Generalized-Tree-Pattern Queries over XML Documents. In *Proc. VLDB*, pages 283–294, 2006.
- [CVZ⁺02] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proc. VLDB*, pages 263–274, 2002.
- [FHM⁺05] M. Fernández, J. Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. Optimizing Sorting and Duplicate Elimination. In *Proc. DEXA*, pages 554–563, 2005.
- [FJSY05] Marcus Fontoura, Vanja Josifovski, Eugene J. Shekita, and Beverly Yang. Optimizing Cursor Movement in Holistic Twig Joins. In *Proc. CIKM*, pages 784–791, 2005.

- [FMM⁺04] M. Fernández, A. Malhotra, J. March, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model. W3C Recommendation, 2004. <http://www.w3.org/TR/xpath-datamodel/>.
- [GD87] Goetz Graefe and David J. DeWitt. The EXODUS Optimizer Generator. In *Proc. SIGMOD*, pages 160–172, 1987.
- [GM93] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. ICDE*, pages 209–218, 1993.
- [Gra94] Goetz Graefe. Volcano—An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.
- [HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible Query Processing in Starburst. In *Proc. SIGMOD*, pages 377–388, 1989.
- [HH07] Michael P. Haustein and Theo Härder. An Efficient Infrastructure for Native transactional XML Processing. *Data and Knowledge Engineering*, 61(3):500–523, 2007.
- [KD99] Navin Kabra and David J. DeWitt. OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. *VLDB Journal*, 8(1):55–78, 1999.
- [Mat07] Christian Mathis. Extending a Tuple-Based XPath Algebra to Enhance Evaluation Flexibility. *Computer Science – Research and Development*, 21(3):147–164, 2007.
- [Mat09] Christian Mathis. *Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems*. Doctoral Thesis, University of Kaiserslautern, July 2009.
- [MH06] Christian Mathis and Theo Härder. Hash-Based Structural Join Algorithms. In *Proc. EDBT Workshops*, pages 136–149, 2006.
- [MHH06] Christian Mathis, Theo Härder, and Michael Haustein. Locking-Aware Structural Join Operators for XML Query Processing. In *Proc. SIGMOD*, pages 467–478, 2006.
- [Mit95] Berhnhard Mitschang. *Anfrageverarbeitung in Datenbanksystemen (Entwurfs- und Implementierungskonzepte)*. Vieweg, 1995. German only.
- [MKL88] Guy M. Lohman Mavis K. Lee, Johann Christoph Freytag. Implementing an Interpreter for Functional Rules in a Query Optimizer. In *Proc. VLDB*, pages 18–229, 1988.
- [MWHH08] Christian Mathis, Andreas Weiner, Theo Härder, and Caesar Ralf Franz Hoppen. XTC-cmp: XQuery Compilation on XTC. In *Proc. VLDB*, pages 1400–1403, 2008.
- [PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. *SIGMOD Record*, 21(2):39–48, 1992.
- [SWK⁺02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, pages 974–985, 2002.

Industrieprogramm

An Integrated Data Management Approach to Manage Health Care Data ^{*}

Diogo Guerra¹, Ute Gawlick², Pedro Bizarro¹, Dieter Gawlick³

CISUC/University of Coimbra¹, University of Utah Health Sciences Center²,
Oracle Corporation³

deag@student.dei.uc.pt, ute.gawlick@hsc.utah.edu,
bizarro@dei.uc.pt, dieter.gawlick@oracle.com

Abstract: Surgical Intensive Care Unit data management systems suffer from three problems: data and meta-data are spread out in different systems, there is a high rate of false positives, and data mining predictions are not presented in a timely manner to health care staff. These problems lead to missed opportunities for data analysis, alert fatigue and reactive, instead of proactive analysis. In this demo, and in contrast to current CEP efforts, we present a proof-of-concept, integrated engine that runs entirely within a single database system. The resulting novel and low cost event processing architecture uses features and components commercially available from Oracle Corporation. We demonstrate how multiple data from a real-world surgical intensive care unit (bed-side sensors and all other information available about the patients) are assimilated and queries, alarms, and rules are applied. The system is highly customizable: staff can point and click to create, edit and delete rules, compose personal rules (per patient, per doctor, per patient-doctor), and, while maintaining a hierarchy of rules, create rules that inherit and override previous rules. The system is also integrated with the data mining module, being able to offer predictions of high risk situations in real-time (e.g., predictions of cardiac arrests). Using simulated inputs, we show the complete system working, including writing and editing rules, triggering simple alerts, prediction of cardiac arrests, and visual explanation of predictions.

1 Introduction

Modern medical institutions have electronic devices that continuously monitor the vital signs of patients such as heart rate, cardiac rhythm, blood pressure, oxygen saturation and many others. Those devices are usually set to trigger alerts for critical values that are above or below predefined thresholds. Due to the static configuration of those thresholds, the devices alert the doctors and nurses for the same critical values regardless of patient condition, demographics, and alarm history. That is, the monitoring middleware typically does not distinguish between:

- A patient with a cardiac condition and a patient without a cardiac condition;

^{*} A shorter version of this paper has been published at DEBS2009. The presentation at BTW 2011 discusses the work described in this paper as well as the follow on work presented at HealthInf 2011 [6]

- A male baby with high heart rate and a female senior with high heart rate;
- A patient that started to have high fever and a patient that has had high fever more than 30 minutes.

Ignoring those differences leads to missed alarms, a very high rate of false alarms, alert fatigue and causes medical staff to ignore most of the alerts.

Another problem is that patient information is normally spread out in multiple independent systems. Those systems contain details of exams such as x-rays, MRI's, blood tests or unstructured free text entered by nurses and doctors. The dispersion of information through independent systems does not allow the systems to make decisions considering the complete patient information.

Data mining systems can detect patterns to predict or identify serious conditions but are normally used offline and in different systems. ICUs are time critical environments where data mining operations (scoring) to identify possible risk situations must be performed continuously.

1.1 Contributions

The goal of this prototype, based on the Surgical Intensive Care Unit environment of the University of Utah Health Sciences Center, was to build a system with the following characteristics:

- A single integrated persistent system to manage historical data, events, rules, and data mining models. See Section 2.
- A highly customizable system that: lets users edit and create rules, maintains a hierarchy of rules, and allows personal rules and complex composable rules, thus contributing to reduce alert fatigue. See Section 3.
- A system able to identify possible future risks by performing data mining in soft-real-time. See Section 4.

2 Components and Architecture

This prototype was developed with Oracle technologies. Some of the technologies used in this prototype (Total Recall, DCN, Rules Manager, Data Mining) have been developed as independent features and not as components of an integrated system for event processing. Part of the challenge and motivation to build this demo was using these technologies to build an integrated system with the characteristics identified above. These technologies and their contribution for the presented prototype are described below.

2.1 Oracle Total Recall (TR)

In a normal database, if you want to store the current and historical values of a sensor, you normally use application logic and write the values time stamped, and possibly in different tables for the actual and historical values. Thus, a sensor with 100 readings takes 100 records in the table(s).

However, using Oracle TR [3] (a technology developed to handle the versioning of records), the reading will take just one record in the table and the other 99 historical values will be transparently kept elsewhere. Those historical versions can be accessed with the AS OF and VERSIONS BETWEEN clauses. The AS OF clause is used to select values in some point in time:

```
-- Select temperature 5 mins ago
SELECT temp FROM patients
  AS OF TIMESTAMP
    systimestamp - INTERVAL '5' MINUTE);
```

The VERSIONS BETWEEN clause queries past ranges:

```
-- Select temp of patient in the last hour
SELECT temp FROM patients
  VERSIONS BETWEEN TIMESTAMP
    (systimestamp - INTERVAL '1' HOUR) AND
    systimestamp
WHERE pid = 10;
```

With TR a slightly extended SQL can be used to access the history; the maintenance of history records is transparent.

2.2 Continuous Query Notification (CQN)

Oracle CQN [2] is a technology that allows the database engine to notify clients about new, changed or deleted data. CQN is different from database triggers. While triggers fire when SQL statements are executed, CQN only notifies about committed data; in one case you see dirty data in the other case you see *committed* data. This technology has two major modes of operation: Object Change Notification and Query Result Change Notification. Query Result Change Notification (the option used in this prototype) allows the user to define with a query what changes should be notified. For example, the user can use CQN to watch for changes in the temperature of patient number 10:

```
SELECT temp FROM patients WHERE pid = 10;
```

CQN will notify only if the temperature of patient number 10 is different at the end of the transaction.

2.3 Oracle Rules Manager (RM)

Oracle RM [1] is a rules engine inside the database. It works based on events that are represented by objects and matches those events to rules defined by the user. The user can define rules that identify, e.g., sequence of events, patterns based on aggregations over time- or count-based windows, non-occurrence of events, disjunction of conditions, or using user-defined functions. RM can watch for changes in tables itself or, as done here, can accept events through its API (accessible from PL/SQL or externally using JDBC and similar interfaces).

RM can define highly complex scenarios without defining very complex rules by using user defined callbacks, production of events to be caught by other rules (see Section 3.1), user-defined conflict resolution policies and user-defined custom event consumption policies. All these features make the system very customizable.

The rules are stored in a table (simplifying rule maintenance) and defined either as a WHERE clause or in a XML specific language as show in Figure 4.

2.4 Oracle Data Mining (ODM)

Oracle DM [4] is data mining engine embedded in the Oracle Database. The users can run data mining algorithms and also build and run models over the data all within the database.

In this project, the data mining algorithms take as input the patient laboratory information and output the predicted risk of patients having a cardiac arrest or respiratory failure in the next 24 hours. The two models, built by Pablo Tamayo from Oracle/MIT, use a 725 patient training set from the SICU of the University of Utah Health Sciences Center.

2.5 Architecture

The goal of the prototype was to develop a single, integrated system able to perform different kinds of processing in a centralized and integrated system.

All the information persists in database tables. These tables are TR enabled and automatically keep history of changes for each record (See 1 in Figure 1). CQN monitors changes on sensor states (2 in Figure 1), generates corresponding events objects and sends them to be consumed by RM by calling its `add_event` function (3 in Figure 1). Then, the RM evaluates the rules and triggers actions to alert doctors and nurses through dashboards and mobile device communication channels (4 in Figure 1). Some rules also trigger calls to data mining models (5 in Figure 1) to apply predictions in real time. The results are sent back to the Rules Manager (6 in Figure 1), which then evaluates new rules to determine if those results need to be pushed to medical staff as well. Thus the data mining operations will only be applied in situations that will likely give some interesting results and not every time the data changes. As described next in Section 3,

the alert system is highly customizable: medical staff can create, edit rules, and delete and even create new rules that inherit information and override other existing rules.

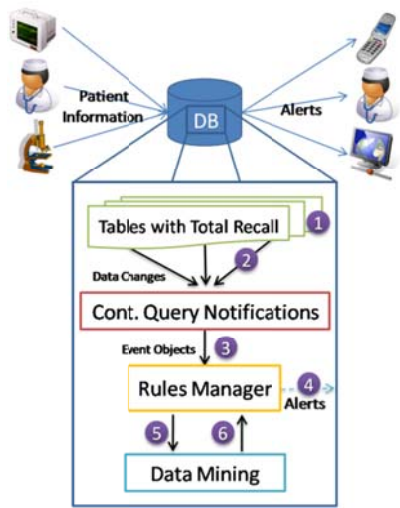


Fig. 1 - Architecture

3 Customization

The customization of rules was a must-have requirement, made possible by Rules Manager. In the alerts section a physician can edit, add or delete alerts. The alerts have multiple priorities: can be applied to all patients, only for one patient, for all patients of a specific physician or only for a specific patient and physician pair. The alerts can relate multiple values, can watch for a state during a specified time, and can also be set to not alert during some time after the first alert to reduce alert fatigue. In Figure 2, the physician overrides a general rule for Sodium level in the guarded priority with a new range. This alert will only trigger for this patient under the supervision of this physician.

For the basic vitals, the ones that can be most frequently needed for alerts, the physician can change their alerts easily with help of sliders to define ranges. This alert will only apply to that patient and for the physician that changed the alert values. To help determine the most appropriate values to the patient, a chart with the history of that vital will appear as shown in Figure 3.

Group (Tip: leave n blank to new group)

4

Priority

Guarded

Type

Patient under Physician

Delay (Tip: delay format is dd hh:mm:ss)

0 12:00:00.00

Description

Abg NA Ute

Action

Change IVF ?

Overrides the Following rules:

Id	Priority	Description
11	Guarded	Abg: HCO3 level guarded

Search Rules to override:

Condition (use &t; and > for = and > respectively)

```
<condition>
  <object name="rt"> rtype = 'ABG_HCO3' AND ((value &t; 18 AND value &t; 22) OR (value &t; 26 AND value &t; 30)) </object>
</condition>
```

Save Cancel

Fig. 2 – Form to edit / add alerts

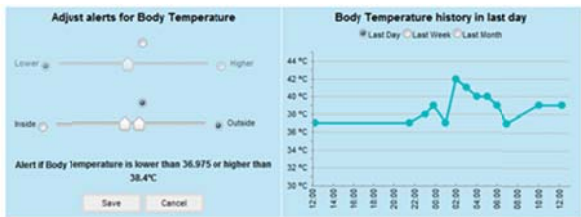


Fig. 3 - Definition of alerts for vitals

3.1 Composability of Rules

The system success depends mostly on the way how rules can be expressed and how complex scenarios can be implemented. Rules Manager has a great support to design complex scenarios based on simple rules and patterns.

The rules are written in XML and allow relating events and performing aggregations over windows and also specifying situations of non events. Many rules could be defined with the language, but is beyond the scope for this demonstration. As example, the rule that checks for an occurrence of a heart rate higher than 120 for more than 1 hour and also that the bp is higher than 80 in the last hour is:

```

<condition>
<and>
  <object name="r0">type = 'heart_rate' and value > 120 </object>
  <notany by="r0.rim$crtime+(1/24)">
    <object name="r0">type = 'heart_rate' and value < 120 </object>
    <object name="r0">type = 'bp' and value < 80 </object>
  </notany>
</and>
</condition>

```

Fig. 4 - Rule example

By composing rules, it is possible to define more complex scenarios than the ones that the language allows in a simple rule. Figure 5 shows a piece of a scenario description in medical technical language to identify possible cardiac arrest situations. The complete rule takes more than a page to be described in technical English and can be implemented in Rules Manager with about 50 rules.

- a. *Persistent or rapidly progressing hypotension*
 - i. MAP, SBP or DBP in serious/critical range >30min
 - ii. MAP, SBP or DBP change from serious to critical
- (...)
- b. *Arrhythmia*
 - i. Asystole (AYST)
 - ii. Supraventricular tachycardia (SVT)
 - iii. Arterial Fibrillation (AFIB)
 - iv. Arterial Flutter (AFLUT)
 - v. VFIB (Ventricular Fibrillation)
- c. *GCS ≤ 8*
- d. *Critical low or high temperature*
- e. *Trauma or Cardiothoracic (CT) patient*

Fig. 5 – Technical definition of a complex rule

The colors represent different weights and a complex alert should occur when one red or three orange or five black statements are true. This is possible using an event approach where each statement is implemented as a simple rule that when is evaluated as true inserts a signaling event into the system with its priority (color). Other rule will be monitoring for those types of events and applies the general rule.

4 Integration with Data Mining

Another feature of the system is the integration of data mining models that score the status of the patients for the probability of risk situations like Cardiac Arrest or Respiratory Failure. These models can be triggered with specific conditions that usually are the main factors or can also be triggered with any change of the values used by the model.

Besides the probability of the risk situation occurs, the physician can also consult the factors that contributed for that prediction. In Figure 6 is an example of a prediction of a Cardiac Arrest with 96% of probability. The physician can look at the chart and see that the negative fractions are values that are normal, or don't contribute for the condition of Cardiac Arrest and the positive ones are the ones that contribute to the cardiac arrest.

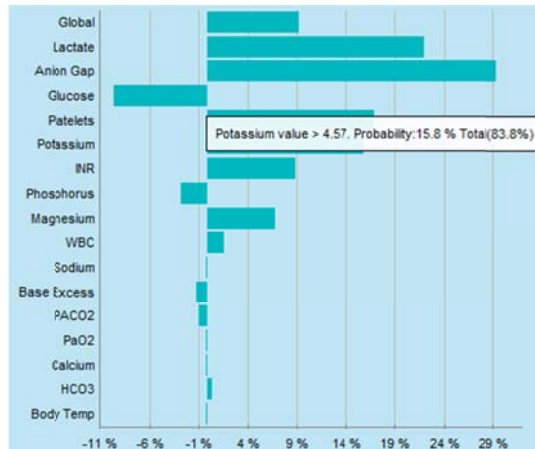


Fig. 6 - Prediction explanation

5 Proof of Concept and Demo

The main goals of this prototype from a medical point of view are: customizable alerts for each patient, reducing alert fatigue, predictive alerting and data mining prediction and evolution. The application developed allows to:

- Monitor the patient vitals, labs and information;
- Easily define custom ranges for alerts for any combination of patient, patient-doctor, or doctor;
- Query patient information history;
- Reduce alert fatigue;
- Support predictions using data mining models.

5.1 Development

The development of this prototype took 6 months from preparing the use case, through the design and implementation of the architecture, definition of the structure and flow of rules manager classes and rules, and development of the web interface Figure 7.

After the understanding of the architecture, the implementation reaches the major complexity implementing the support for the use case inside RM. The definition of consumption rules, callback functions for rules, cascading rules through generation of control events.

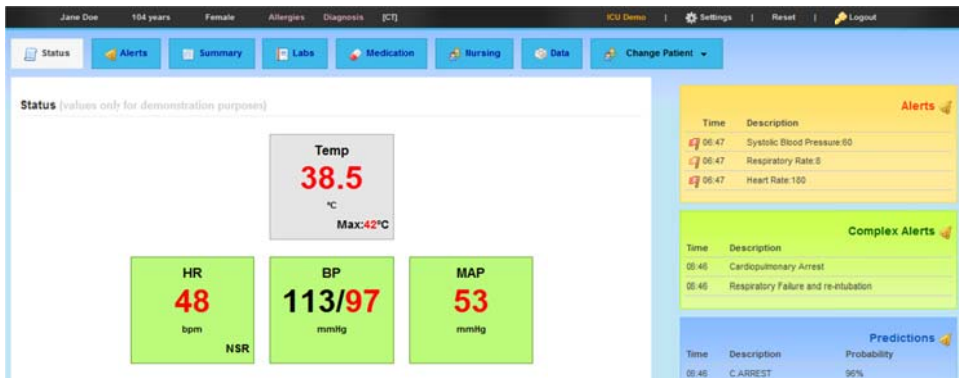


Fig. 7 - Interface

5.2 Demonstration

In this demo the main features of the system will be demonstrated. The users can update data, watch the alerts being triggered depending on the patient or doctor, define specific rules, being alerted from data mining models when high risk situations may occur. The use case is discussed elsewhere [5].

Acknowledgement: This work was supported to a great extent by Oracle staff: Aravind Yalamanchi, Pablo Tamaya, Srinivas Vemuri and Venkatesh Radhakrishnan. Also Dr. Sean Mulvihill, Dr. Edward Kimball, and Dr. Jeffrey Lin at University of Utah Health Sciences Center.

6 References

- [1] Oracle® Database Rules Manager and Expression Filter Developer's Guide 11g Release 1 (11.1). Available at: http://download.oracle.com/docs/cd/B28359_01/appdev.111/b31088/toc.htm, accessed May 2009.
- [2] Oracle® Database PL/SQL Packages and Types Reference 11g Release 1 (11.1). Available at: http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28419/d_cqnotif.htm, accessed May 2009.
- [3] Oracle® Database Advanced Application Developer's Guide 11g Release 1 (11.1). Available at: http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28424/adfns_flashback.htm, accessed May 2009.
- [4] Oracle® Database PL/SQL Packages and Types Reference 11g Release 1 (11.1). Available at: http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28419/d_datmin.htm, accessed May 2009.

- [5] P. Bizarro, D. Gawlick, T. Paulus, H. Reed, M. Cooper. Event Processing Use Cases Tutorial. DEBS'2009.
- [6] D. Gawlick, Adel Ghoneimy, Zhen Hua Liu. How to Build a Modern Patient Care Application. To be published at HealthInf 2011.

Involving Business Users in the Design of Complex Event Processing Systems¹

Jens SCHIMMELPFENNIG, Dirk MAYER, Philipp WALTER, Christian SEEL

Research Projects Dept.

IDS Scheer AG

Altenkesseler Str. 17

66115 Saarbrücken, Germany

{ jens.schimmelpfennig | dirk.mayer | philipp.walter | christian.seel } @ids-scheer.com

Abstract: Complex Event Processing (CEP) gained more and more attention in research and practical usage during the last years. Several engines and languages have been created and adapted in order to handle complex events. To describe the aggregation of simple events to more complex ones, these engines use technical and formal languages which are difficult to handle for business experts. Hence, business experts are often not properly involved in the definition of complex events. This lack in the area of requirement engineering worsens the result of many CEP projects. Therefore this paper proposes a conceptual modeling language that allows business experts to specify complex events according to actual business needs. The proposed modeling language is demonstrated in a quality management scenario in the automotive manufacturing industry and is defined with OMG's MOF standard.

¹ This paper is based on our work in the research project ADiWa, which is funded by the German Ministry for Education and Research (BMBF) under funding reference number 01IA08006.

1 Introduction

Within the last years, complex event processing technology continuously matured and is going to leave its technical niche. The founding of the Event Processing Technical Society (EPTS), their aim to provide a foundation for analysis and communication [1] and the fact that nearly all big business software vendors extend their portfolios with a complex event processing (CEP) engine indicate that CEP is becoming more important for daily business. Apart from that, the possibility to fit more and more prefabricated parts or products with RFID tags in an affordable manner will also steadily increase the amount of real-time information that must be processed in an enterprise. Since these isolated pieces of information about one component are usually not relevant for business decisions, they must be aggregated and put into a larger context to allow for supporting business-level decisions. Hence the aggregation of single events to complex and business relevant ones with the help of CEP engines [2], [3] is coming to the fore. CEP engines allow aggregating several basic events to complex ones and provide a history of both types of events. Therefore complex events can be generated that trigger specific business processes or even invoke changes in a running process instance. For example the complex event “15 percent of parts from the same charge assembled within one hour have a different weight than expected” can trigger both a business process for removing the entire charge from production and a supply chain process to order new charges as reimbursement. Despite their relevance for business processes, complex events are commonly represented by query languages or XPath expressions [4].

These languages are well comprehensible for technical staff, so implementing a precisely specified CEP query is normally not much of a problem. However, the necessity for CEP usually originates from a business scenario that mostly non-technical domain experts with an economic background are familiar with, such as the stock manager in a manufacturing company. I.e., successful CEP applications require the involvement of business domain experts who usually lack of profound technical knowledge in complex event processing languages required for the specification of complex events. So from a technical point of view, the challenge is to allow business people to specify their CEP requirements in a standardized and technically sound way.

To bridge this gap between business and IT view and to involve business users and process owners in CEP implementations, we propose a conceptual semi-formal and graphical modeling language that enables users to specify complex events in a business-oriented manner. The language allows for using various data sources in order to describe complex events and the aggregation of basic and complex events to (higher) complex events in any number of aggregation levels. A technician can transform such conceptual CEP models more precisely and easily into technical CEP specifications than, requirements written down in textual form. Furthermore, complex events are required for business process modeling in order to describe reactions to complex event occurrences as business processes. After setting the basic principles, we demonstrate our approach using a production process from an automotive scenario as an example. The paper closes with a summary and provides an outlook on further research requirements.

2 Theoretical Foundations

2.1 Methodology Used

In an endeavour to contribute to the field of information system (IS) research, this paper follows the seven guidelines for design science in IS research by Hevner et al. [5]. Especially the guidelines of relevance and of the creation of an artifact as result of the design science process have been followed. The relevance of the research topic was derived from requirements the authors were confronted with during the research project ADiWa, funded by the German Ministry of Research and Education (BMBF), and in several industry consulting projects. The resulting artefact of the research process is a modeling tool and its underlying concept. Moreover, Weick's sense making paradigm [6] is taken into account which ensures the consistency of the developed modeling language and its integration into business process modeling approaches.

2.2 Terminology and Related Work

Before our complex event modeling concept is presented, the underlying terminology is briefly clarified in this section. However, tracing former and current terminological discussions would exceed the scope of this article, so we confine ourselves to the characterization of the terms "event", "complex event" and "complex event processing".

An event is considered an abstracted data representation of a real-world happening. Thus it is associated with that point in time when the underlying real-world incident happened. The event's data structure highly depends on the scenario and real-world happening that is to be represented. Usually, event modeling aims for lightweight events that can be quickly transferred and processed in large amounts. Events are immutable, i.e. they should never be altered once they are created. Events are temporary, i.e. they are rather intended for real-time or short-term processing instead of long-term storage and ex-post evaluation. They are neither intended to replace the operational control flow (which remains subject of established technologies like EAI), nor as a means of point-to-point communication. Normally they are generated, broadcast and processed by a large number of application systems simultaneously, thus forming a so-called event cloud comprising thousands of events.

Although generating events is a simple and fast procedure for every single system participating in the event cloud, they allow to reconstruct a situational "big picture" across an arbitrary number of systems by identifying sets of interdependent events. If a situation arises that is identified by a characteristic sequence of events, a new event is generated which announces this situation. Since the event is a summary of many "simple" events, it is called a complex event. The process of aggregating a set of events according to a predefined rule describing such a situation is called complex event processing (CEP).

A popular example for the application of CEP is credit card fraud detection [7]. It is presupposed that for every credit card transaction, the banking IT generates an event including the amount, the beneficiary, the card number and the location of card usage. A specialized monitoring system then scans this “event noise” for patterns that indicate credit card fraud: e. g. when a card is used in New York at 3 pm and in Paris at 4 pm, this strongly indicates fraud since no one can travel that fast from New York to Paris.

From a general implementation point of view, there is a multitude of standards and technologies available for developing CEP solutions which can be distinguished in two major paradigms. Stream-based approaches usually implement an SQL-like language to inspect and modify event streams [8], e. g. Esper [9], SASE [10] and StreamSQL [11]. Rule-based approaches allow specifying ECA rules for event processing, e. g. JBoss Drools [12].

Open source and commercial frameworks, emerging and mature solutions, simple and complex packages – even a concise review of the available software would go beyond the scope of this article. Therefore we do not presume a specific CEP technology to build our conceptual modeling approach upon. Instead, the proposed modeling method aims at abstracting from the underlying CEP technology. Of course this implies that the expressiveness of the conceptual layer must be adaptable to the technical realization – however, this is beyond the scope of this article and will be treated in a later publication.

3 Modeling of Complex Events

3.1 Application Scenario

To outline the benefits and to exemplify the usage of conceptual modeling of complex events, we chose an assembly line production process in the automotive sector. Within the assembly line, each future car is passing through different stations where delivered preassembled components are installed. According to lean and just-in-time production, these components are delivered exactly at the right time and in the right amount to the station where they are assembled. For each incoming vehicle, the assembly worker at a station is instructed which parts to assemble. The parts are identified by RFID tags to ensure that the correct variant for a certain vehicle is assembled.

Today, following Kaizen [13] and total quality management principles, each employee working on the assembly line can stop the line by signalling a problem – we will furthermore call this an “incident report”. A responsible head to the station actuates the stop, checks and analyzes the problem and resumes production after the problem is solved. Beyond that, there is an additional final inspection of each vehicle produced which may also reveal quality defects unnoticed during the assembly.

This procedure is considered an effective and efficient reaction to emergent problems as they are handled “right at the source” whenever possible. However, subtle increases of incidents and/or quality defect frequency often remain unnoticed for some time, although they may indicate a larger underlying problem, e. g. a defective lot of supplied parts. Hence it would be very valuable to detect such problems at an early stage. Unfortunately, recognizing such problems requires complex considerations of a lot of factors and thus could not be modeled in an intuitive manner. Within the next sections, we describe a modeling concept to deal with such problems using the example given above. First we exemplify the conceptual modeling layer and then we shortly outline the technical realization.

3.2 Conceptual Modeling

In current business process management approaches, processes are documented using semi-formal modeling methods. The Event-driven Process Chain (EPC) [14] is a widely accepted standard in process modeling. Here, business relevant events and corresponding business functions are put into sequence in order to define the behaviour of a business process. A process model thus defines a sequence of actions which are triggered by certain events and which produce certain events. However, this linear control flow with its strong emphasis on business functions makes it difficult to express complex situations such as a slow increase in incidents exemplified above. It is not necessarily impossible to express such conditions in an EPC, but the model complexity would be immense and reduce the expressiveness and readability of the model. Therefore we propose to introduce a new modeling language called Event Structure Model (ESM) which supplements the EPC. It allows defining situations by specifying certain event constellations and how to react to such situations by generating new events. Thus an ESM model can connect several EPCs by collecting relevant events from different EPCs and by producing new events triggering other EPCs.

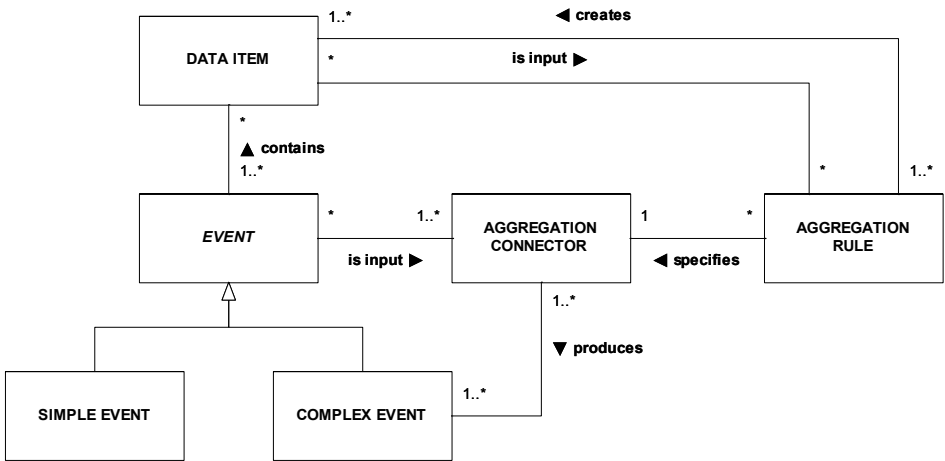


Figure 1: Meta model of the Event Structure Model (ESM)

In order to describe ESM we use OMG’s Meta Object Facility (MOF) [15]. MOF uses UML class diagrams that can be enhanced with Object Constraint Language (OCL) statements. Figure 1 shows the meta model of the ESM language.

The entry point of the meta model is the abstract class “Event”. Events can be distinguished in simple events and complex events. Complex events can be constructed from an arbitrary number of simple or other complex events. The aggregation connector creates the relationship between complex events and events that it is deduced from. The way how events are aggregated to a complex event is described as an aggregation rule, which is attached to each aggregation connector. The aggregation rule is based on data items that are provided by an event. These data items are used in the aggregation rule in two ways. First, they are used in order to specify whether the complex event occurs. Second, data items of the complex event can be created based on data items of its ingoing events. The graphical notation is presented in the scenario below.

To visualize the functionality of ESM, the application scenario introduced above is further detailed by an exemplary ESM. On the left-hand side of figure 2, a simplified process model of the application scenario is depicted as an EPC. For each vehicle to be assembled, parts are installed at different assembly stations. Once the installation of a specific part succeeded, the vehicle is transported to the next station until the end of production line is reached. A final quality check is conducted, ensuring the overall quality of the product.

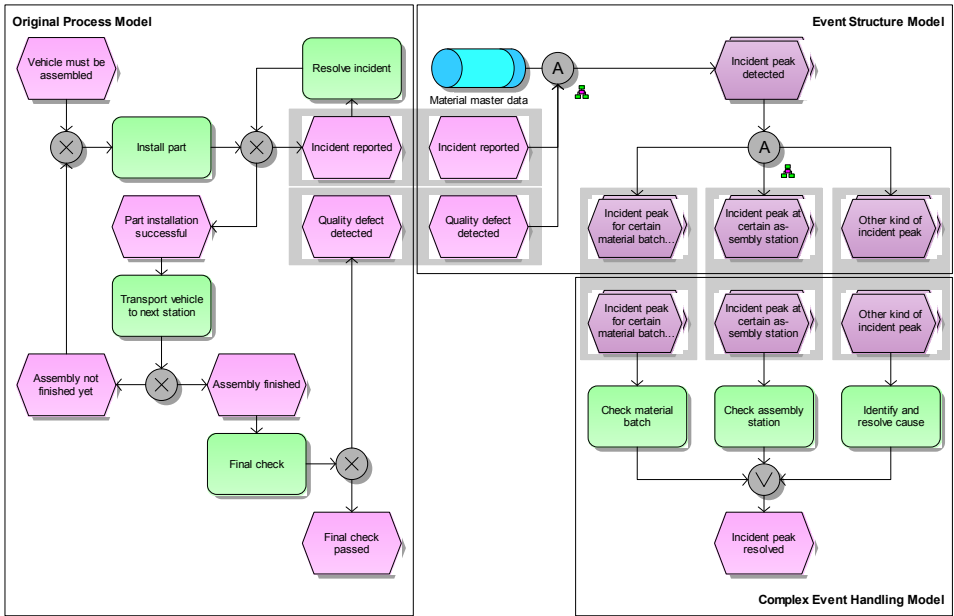


Figure 2: An Event Structure Model (ESM, top right) mediates between the standard production process (left) and the separate process for handling exceptional incident peaks (bottom right)

Incidents in the assembly process cause the assembly worker to stop the production line in order to resolve the issue. At the point of detection, the main priority is to resolve the error as fast as possible to minimize the delay in production, providing very little room for thorough root cause analysis and correlation of incident reports. Similarly, if a defective part is identified in the final quality check, it is communicated by the event “quality defect detected” and repaired with high priority in order to deliver the final product.

Since there may be multiple lines of production within a factory, peaks within the overall number of similar incidents may not be obvious, especially if the incidents happen during different working shifts or at different stations. However, the identification of peaks in related incidents (e. g. caused by defective parts of the same lot) is vital in order to optimize the production process and to prevent incidents from reoccurring due to the same cause. By correlating all incident events occurring in the production process, the complex event “incident peak occurred” can be refined.

In our approach, the structural aggregation of the complex event is specified as an ESM. As depicted in the top right corner of figure 2, complex events are aggregated by connecting member events from process models as well as external information carriers using a complex event aggregation connector (“A”). Each of the ingoing objects is typed by an assigned data type model, representing the “payload” that is communicated by those events or information carriers. The aggregated complex event itself can be used to define further business relevant complex events. In the application scenario, the two events “incident reported” and “quality defect detected” as well as the part’s material master data are used to form the complex event “incident peak occurred”. The latter is used to deduce three business relevant events which serve as trigger for optimization processes as shown in the lower right corner of figure 2.

Once the structural aggregation is defined, a mapping of ingoing event data to the complex event’s data structure is to be specified. Within that mapping, arithmetic and logical operators are available in order to define the desired payload of the resulting complex event.

By defining complex events in the ESM using simple events from process models, the conceptual aggregation aspects are covered. For the deployment of complex event definitions, technical aspects such as data mapping and filtering of data streams need to be covered. These technical modeling aspects are assigned to the complex event aggregation connectors and are described in detail in the next section.

3.3 Technical Modeling

When creating ESMs, it must be taken into account that the majority of business relevant events defined in EPC models are of complex nature (such as “goods received”) and are not directly measurable in a company’s runtime environment. From a business user’s view, the information provided is sufficient in terms of business process management; however, such “narrative” declarations lack execution-relevant information on the actual process implementation within the company.

To allow for an implementation of the ESM described above, it is necessary to amend it with technical details that specify the event aggregation operationally. For that purpose, platform independent technical models are added to the conceptual model. Actually every A-connector in the conceptual ESM is assigned with such a technical aggregation rule. As an example, figure 3 shows the technical model for the upper left A-connector in the ESM in figure 2.

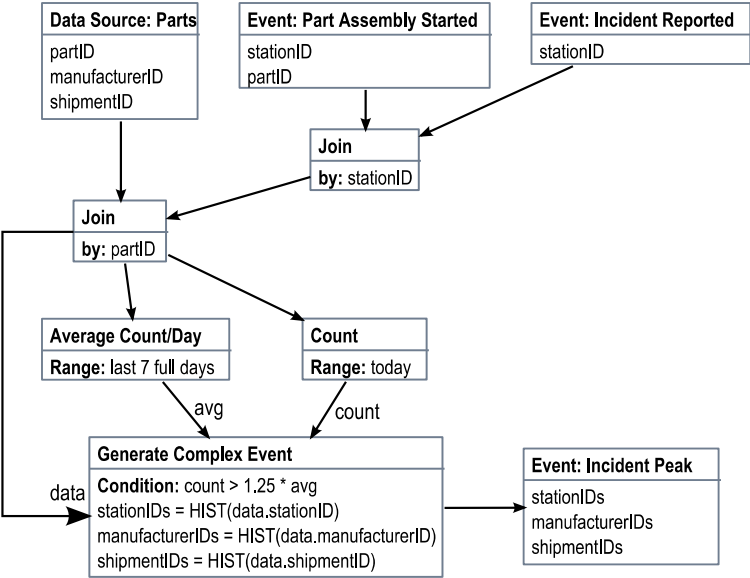


Figure 3: Technical CEP model describing the generation of the complex event “incident peak detected”

The technical model is laid out from top to bottom. At the top there are the information sources, i.e. the material database and three types of events. They are used to construct a single event stream consisting of incident and quality defect reports exclusively. Construction starts with an “incident report” event only carrying the number of the station reporting the incident. However, this information is too scarce to analyze the cause of a peak later, so it is combined with the previous event “part assembly started”. Each part is equipped with an RFID tag and whenever an employee starts assembling a part, he first scans its tag. Every such RFID scan generates a single event “part assembly started” which carries the scanned part number. If an incident is reported afterwards, this part is assumed to cause the incident and its number is added to the incident report event by joining it with the previous assembly start event. In the conceptual model, this connection to the assembly start event is not expressed since it is more a technical than a conceptual necessity.

In a similar manner, quality defects detected during the final quality assurance (QA) check are broadcasted as events. The quality defect report names the objected parts, so their numbers are included in the QA events. The station ID of the QA events is set to the QA station number, so the QA event structure is exactly the same as the incident report event structure: both have a station and a part number. Thus the origin of such an event is no longer of concern; both types are furthermore treated as „incident report events“.

Similarly to joining event streams it is possible to include static data in events. The join on the left adds additional material data (manufacturer and shipment ID) from the material database to the events. Finally there is an incident report event identifying station, part, manufacturer and shipment related to the incident.

In the next step the stream of these events is analyzed: the average incident frequency within the last full seven days is determined and is compared to the number of today's incidents. If the number of today's incidents exceeds 125% of last week's average, the complex event "incident peak detected" is generated. It is amended with information about the statistical distribution of stations, manufacturers and shipments involved in these incidents.

The technical model of the second A-connector isn't shown here since it is quite simple: if the station or the shipment distribution shows a significant accumulation of a single ID, the incident peak is likely to be caused there and must be investigated accordingly.

4 Conclusion and Outlook

In this paper, we propose an extension to the Event-driven Process Chain (EPC) which is called Event Structure Model (ESM). On a conceptual level, ESM allows to associate different EPC models by specifying conditional transitions from one set of events to another one, thus defining processes which react to certain situations. On a technical level, ESM models can be amended with precise aggregation rules which allow their realization using Complex Event Processing (CEP) engines. However, the distinction between conceptual and technical layer allows separating the "business logic" from its realization so that CEP technology becomes accessible to non-technical business experts. Besides specifying ESM formally, we exemplified its usage in an application scenario in the automotive sector.

There are mainly two points of contact for subsequent research. First, on a conceptual level, other application scenarios that allowed for refining ESM, from industrial as well as non-industrial branches, would be a valuable addition to this work. Second, from a technical point of view, ensuring compatibility of different CEP engine concepts with ESM still poses a challenge. Although both points will be addressed in the context of our further research efforts, a lively scientific discussion with diverse contributions could show up new and important aspects of conceptual event modeling.

References

- [1] Luckham D. C, Schulte R.: Event Processing Glossary, 2008. http://www.ep-ts.com/component/option,com_docman/task,doc_download/gid,66/Itemid,84/ , accessed 09/23/2010.
- [2] Luckham D. C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Reading, 2006
- [3] Greiner T., Düster W., Pouatcha F., von Ammon R., Brandl H.-M., Guschakowski D.: Business activity monitoring of norisbank taking the example of the application easyCredit and the future adoption of complex event processing (CEP). In Proc. Int. Symp. on Principles and Practice of Programming in Java. ACM, New York, 2006, pp. 237-242
- [4] Bry F., Eckert M.: Rule-Based Composite Event Queries: The Language XChangeEQ and its Semantics. In: Web Reasoning and Rule Systems. Lecture Notes in Computer Science, 2007, Volume 4524/2007, pp. 16-30.
- [5] A. Hevner, S. T. March, J. Park, S. Ram: "Design Science in Information Systems Research." MIS Quarterly 28, Nr. 1 (03 2004): pp. 75-105.
- [6] K. E. Weick: Sensemaking in Organizations, Thousand Oaks, CA, USA, 1995.
- [7] Widder, A.; v. Ammon, R.; Schaeffer, P.; Wolff, C.: Identification of suspicious, unknown event patterns in an event cloud. In: ACM International Conference Proceeding Series; Vol. 233: Proceedings of the 2007 inaugural international conference on Distributed event-based systems. Toronto, Ontario, Canada, 2007, pp. 164-170.
- [8] Wu, E.; Diao, Y.; Rizvi, S.: High-performance complex event processing over streams. In: SIGMOD '06 Proceedings of the 2006 ACM SIGMOD international conference on Management of data.
- [9] ESPER: <http://esper.codehaus.org/>, last access 12-13-2010.
- [10] SASE: <http://sase.cs.umass.edu/>, last access 12-13-2010.
- [11] StreamSQL: <http://www.streambase.com/complex-event-processing.htm>, last access 12-13-2010
- [12] JBoss Drools: <http://www.jboss.org/drools/>, last access 12-13-2010.
- [13] Imai, M.: Kaizen. The Key to Japan's Competitive Success. 1986.
- [14] Keller, G.; Nüttgens, M.; Scheer, A.-W.: Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik. A.-W. Scheer (Hrsg.). No. 89, Saarbrücken 1992.
- [15] Object Management Group (OMG): Meta Object Facility (MOF) 2.0. <http://www.omg.org/spec/MOF/2.0/> accessed: 09/03/2010.

Fast and Easy Delivery of Data Mining Insights to Reporting Systems

Ruben Pulido, Christoph Sieb

rpulido@de.ibm.com, christoph.sieb@de.ibm.com

Abstract: During the last decade data mining and predictive tools have evolved to a mature level. Many companies and organizations use those techniques to derive advanced insight from their company data. Today, mining experts analyze the data and manually deliver the results to a rather small number of decision makers. To make data mining insights available to a broader audience, manual delivery is not sufficient anymore. Suitable delivery vehicles are reporting systems, which are well known to business users. However, in most cases there is no straight forward delivery path. Much manual work and knowledge is necessary to make the insights consumable by reporting systems.

In this work we propose a system that allows fast and easy delivery of data mining insights to report servers. The system automatically analyzes and transforms mining results, generates reports and deploys them to the report server.

1 Introduction

Data mining is nowadays widely spread across enterprises, as a technique to derive valuable insight from large amounts of business data. This insight is then used to support the decision making process. In the last decades, research has mostly focused on generating data mining results rather than delivering them to a broad audience in an easy consumable way. For instance, in the retail sector, a mining based customer segmentation, combined with a customized product affinity analysis allows fine grained decision support even for front line employees. The employee can not only focus on the suitable customer group but also on the products and product groups she is responsible for. Most important however, is that employees are provided with this information in an easy consumable way. In most companies the enterprise reporting system represents the most appropriate way delivering those insights. Employees can easily access reports through their web browser and are familiar with the system interaction.

However, bringing this insight from the data-mining-expert-tool to the reporting front end, is not straight forward. Reporting systems usually access data stored in relational tables. Data mining results, however, are very different from flat table structures. They are mostly stored in hierarchical ways in large documents (e.g. standardized PMML [PMM] format). Since most of today's BI systems cannot directly consume data mining models, the mining insight must first be transformed to a form that is consumable by these systems.

In the following section we show related approaches. In section 3, we illustrate, how the

proposed deployment system is used to deliver insightful data mining reports. In section 4 we describe the underlying architecture of the system and in section 5 we shortly outline products which are suitable in the context of the proposed deployment system.

2 Related approaches

Few vendors provide dedicated report design tools in which a report designer can manually create mining insight reports. Vendors, not providing dedicated tools force the user to transform mining insight into a form consumable by the BI system. Further, deep mining knowledge is required to create mining reports with rather general report tools. The creation of such reports is a tedious task and changes in the underlying data result in long lasting manual changes. Further, the task of transforming the mining insight, creating the reports and meta information requires deep knowledge of the involved tools. All those requirements lead to several people across the enterprise being involved.

Known solutions are based on exporting images generated within the mining tool. However, this is a very static and non-interactive way. Further, this solution does not provide automatic deployment of the mining insight. Few tools allow visualizing standardized mining models natively. This approach is however not very flexible since it restricts the visualization of data mining insight to predefined graphics [IBM]. Microstrategy also provides basic native PMML support. However, the reports are assumed to be designed by a report designer [Mic], and not automatically generated by the system.

3 User interaction

The system we propose allows fast and easy delivery of data mining insights into reporting systems. The steps required to make those insights available in the reporting front end are described in this section.

A data mining expert first selects the most appropriate data mining method to solve the business problem he is facing. He then locates the business data available and explores its structure. After preparing the data to be used as input for a data mining method, the data mining expert can choose among an unlimited combination of available algorithms and settings. This is normally an iterative process, in which the data mining expert performs several data transformations and creates multiple data mining models and tests them. The data mining expert then visualizes the data mining models using visualization capabilities of the data mining tool, as shown in figure 1. Once the data mining expert is satisfied with one of the created data mining models, this model is ready to be put into production. It is ready to be integrated as part of an automated business process, or it can be deployed to a reporting system where employees can access it using the BI front end tool.

With the system presented in this article no additional people with reporting skills are needed to create the report metamodel and report specifications. Rather, the data mining

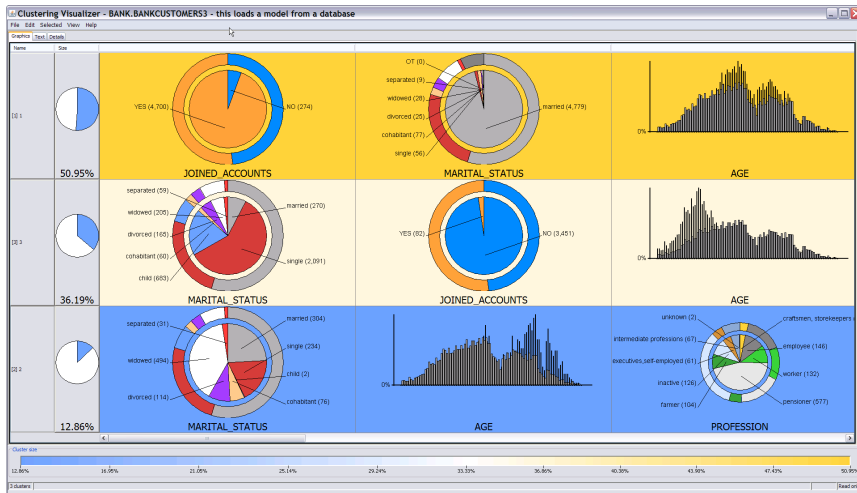


Figure 1: Visualization of a clustering model using the data mining tool

expert can directly deploy the data mining insight by following the steps, illustrated in figure 2.

- Enter the url where the reporting server can be accessed.
- Enter the credentials needed to authenticate against the reporting server.
- Specify the location (i.e. database, schema and table names) on the relational database in which the data mining model is to be extracted.
- Specify the location on the reporting system, on which the report meta model and the report specifications will be generated.

The system then automatically analyzes and transforms the mining results, generates the report metamodel and report specifications and deploys both to the reporting system.

Figure 3 shows one of the reports that are made available through the BI reporting system. In this example the user of the reporting system can visually analyze the content of the data mining insight, describing the different clusters in which the customers of the company have been segmented. Each row of the report represents a certain cluster, and each column represents a certain field or attribute (e.g: age, gender, average balance...). In each bar chart the distribution for a field of the customers belonging to a certain cluster is overlapped to the expected average distribution of all customers in the company. E.g. cluster 3 represents the elderly people who are rather female and have rather bank cards.

The system presented in this work does not only allow visualizing static data mining insight, which resides on the database prior to report execution. It is also capable of generating and deploying reports which allow the end user to invoke data mining directly from

Deploy Mining Model to Cognos

Select Cognos Server URL
Enter the URL where Cognos Server can be reached

Cognos Server URL

[?](#)

(a)

Deploy Mining Model to Cognos

Login to Cognos Server
Select the Namespace and enter your credentials to logon to Cognos Server

Namespace

User

Password

Anonymous Logon Section
☒ Logon anonymously

[?](#)

(b)

Deploy Mining Model to Cognos

Select database schema and tables prefix
Select database schema and prefix name for the tables in which to extract the model information

Schema

Tables prefix

[View created report in browser](#)

☐ View report

[?](#)

(c)

Deploy Mining Model to Cognos

Select Cognos package and Cognos report Prefix
Select the name and location of the Cognos package to be published and the prefix name for the Cognos reports to be generated

Select the folder where to deploy the Cognos package

- My Folders
 - Customer Analysis
 - Customer Segmentation**
 - Product Sales
 - Returned Products Analysis

Package name

Reports prefix

[?](#)

(d)

Figure 2: Steps to deploy data mining insight into reporting tool: (a) Enter the url for the reporting server; (b) Enter the credentials to authenticate against the reporting server; (c) Specify the location on the relational database in which the data mining model is to be extracted; and, (d) Specify the location on the reporting system, on which the reporting meta model and the report specifications will be generated.

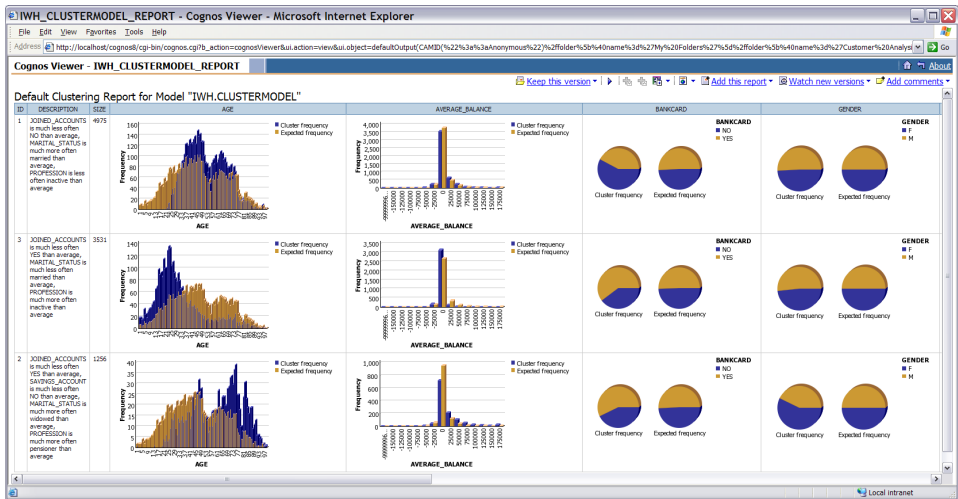


Figure 3: BI report presenting data mining insights for a clustering model

the reporting front end. E.g. the user can select the subset of data to be used for mining or he can configure simple parameters of the data mining algorithm, prior to execution.

Figure 4 shows how the user interacts with a report that allows dynamic invocation of association rule mining on a dedicated cluster selected by the user. The user simply performs the following steps:

- Select a certain cluster of customers, for which he wants to compute association rules.
- Enter a value for the minimum support of the rules to be computed.
- After submitting the report, the association rules are calculated by invoking the In-Database-Mining functionality.
- The calculated rules fulfilling the support threshold set by the user are then presented in the report front end.

The generation and deployment of such a report is made in a similar way as described in figure 2.

4 Architecture

The proposed deployment system links two fundamental BI structures. The Warehouse, based on a relational database that provides In-Database-Mining functionality and the BI system. The general functionality of the deployment system comprises:

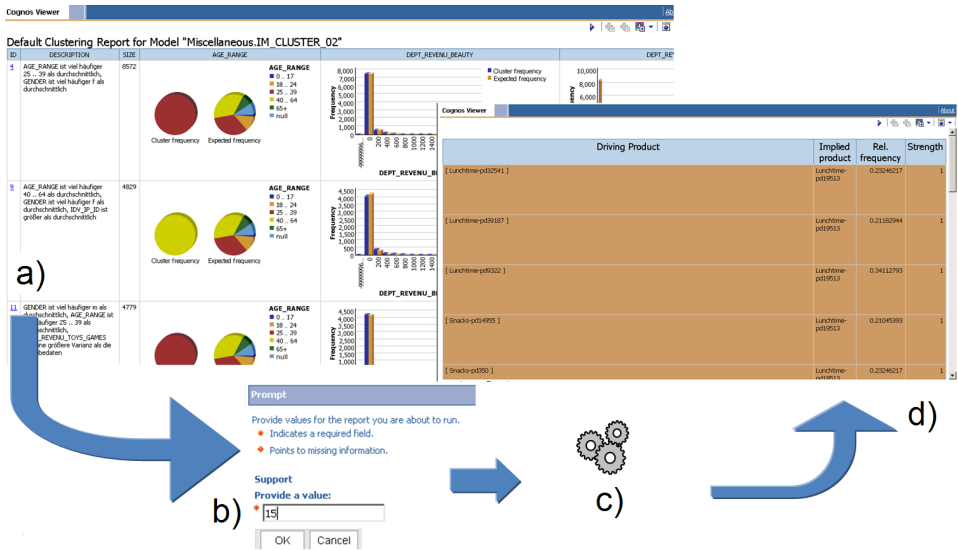


Figure 4: BI report invoking association rules data mining on demand on a subset of the data selected by the user

1. Translating the predictive insight into a common format understandable by the BI system
2. Generating meta information and report specifications for the BI system that are suitable to visualize mining / predictive insight in an appropriate and understandable format
3. Deploying the meta information and reports to the BI system.

There exist two fundamental approaches. A static approach in which a mining expert creates mining models that need to be delivered to the business user. And a dynamic approach, in which the mining expert just defines the process to create the mining model, which can then be invoked dynamically by the user through the BI system. The next chapters describe both approaches in detail.

4.1 Static mining content

The static approach is based on a **Mining Model** (see Figure 5)¹ created by a mining expert (including **Data Preparation** and **Modeling** steps). The deployment system automatically creates a table representation from the **Mining Model** and extracts it to the database as a

¹For the whole paragraph, the bold terms refer to Figure 5

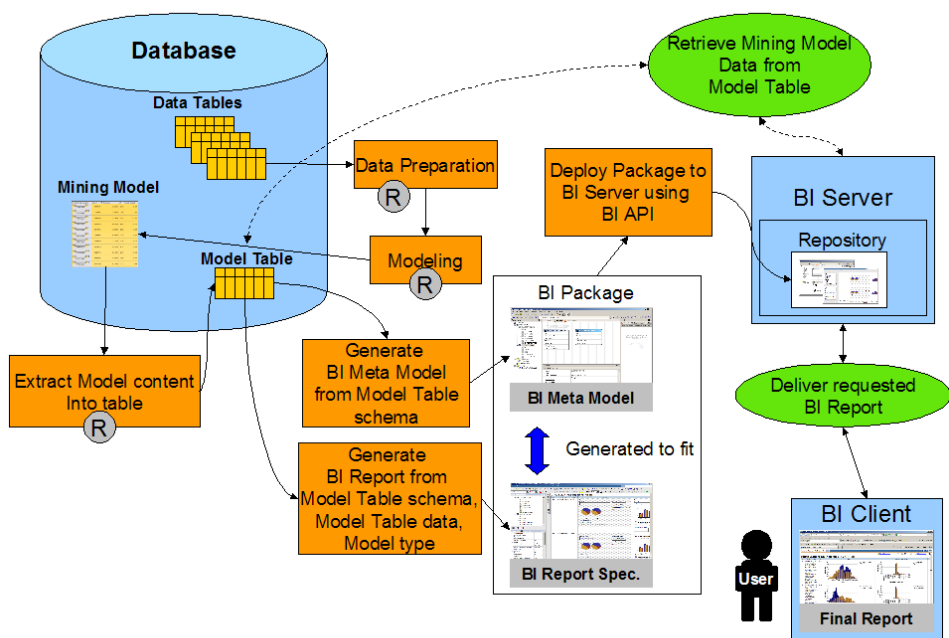


Figure 5: System architecture for deployment of static mining insights

Model Table. This representation is done in a fashion such that the mining insight can be accessed by the **BI Server**.

In a second step, the deployment system generates the necessary meta information (**BI Meta Model**) and **BI Report Specifications** required by the BI system. This meta information and reports are dynamically created based on the content of the **Mining Model** and the layout and data of the **Model Table** containing the mining insight.

Finally, the reports are automatically deployed to the **BI Server**. The deployment system uses the **BI Server's** API interfaces to deploy the generated **BI Meta Model** and **BI Report Specifications** without manual user interaction. It also triggers creation of the actual report from its specification within the **BI Server**. Then, the **User** can access the mining / predictive insight like any other report using the **BI Client**. The **BI Server** retrieves the mining insight directly from the Model Table.

Even though the report specification is static, the actual content can be updated. In more detail, the information contained in the **Model Table** can be updated by re-executing the steps marked with an R in Figure 5. E.g. the re-execution may result in a different clustering reflected by different information in the **Model Table**. The report specification remains the same but the content is retrieved from the updated **Model Table**. This update process

can also be incorporated into automatic business processes to keep the mining content up to date.

Generation of the BI-Meta Model:

The **BI Meta Model** generation is based on templates. The templates contain basic structures for the meta model according to the underlying mining method (Clustering, Classification, Association Mining,...). The deployment system first analyzes the layout of the **Model Table**. From this analysis, the meta model objects are derived. Second, the deployment system analyzes the actual data to derive meta model object types.

Generation of the Report specifications:

The generation of the **BI Report Specifications** is also based on templates. The templates contain basic structures for the report specification depending on the mining model type. During generation, the deployment system analyzes the **Model Table** content. The data of the model table is e.g. analyzed for the number of features and their relevance for each cluster. The report can then be restricted to those relevant features. Furthermore, the forming of the reports and charts is optimized based on the data of the model table. For each model type may exist several report specifications that are linked with each other. E.g. detail reports for dedicated charts or drill through reports as explained in the next paragraph.

Drill through aspect: Often users need to know details from the underlying data from which the data mining model was generated. The deployment system automatically incorporates so called Drill through data into the reports, allowing for better understanding of the mining model. Most useful is drill through data representing typical records. E.g. For clustering, typical records are those which best represent the characteristics of a certain cluster. The deployment system automatically detects such records and incorporates them into the reports.

4.2 Dynamic mining content

The deployment system also generates and deploys dynamic reports which invoke mining at the time the report consumer interacts with the BI system. This allows the user to customize the mining-reports by passing parameters and other settings.

The deployment system automatically creates a **Stored Procedure** (see Figure 6)² from the **Data Preparation** and **Modeling** steps which are defined by a mining expert. The **Extraction** step is also performed within the **Stored Procedure**. The **BI Meta Model** is now based on the stored procedure instead of the **Model Table** allowing for dynamic interaction. The stored procedure can be invoked by the BI Server passing parameters entered by the **User** using the **BI Client**. The dynamically created mining insight is then retrieved by the **BI Server** from the **Stored Procedure** result set.

Generation of the Stored Procedure:

²For the whole paragraph, the bold terms refer to Figure 6

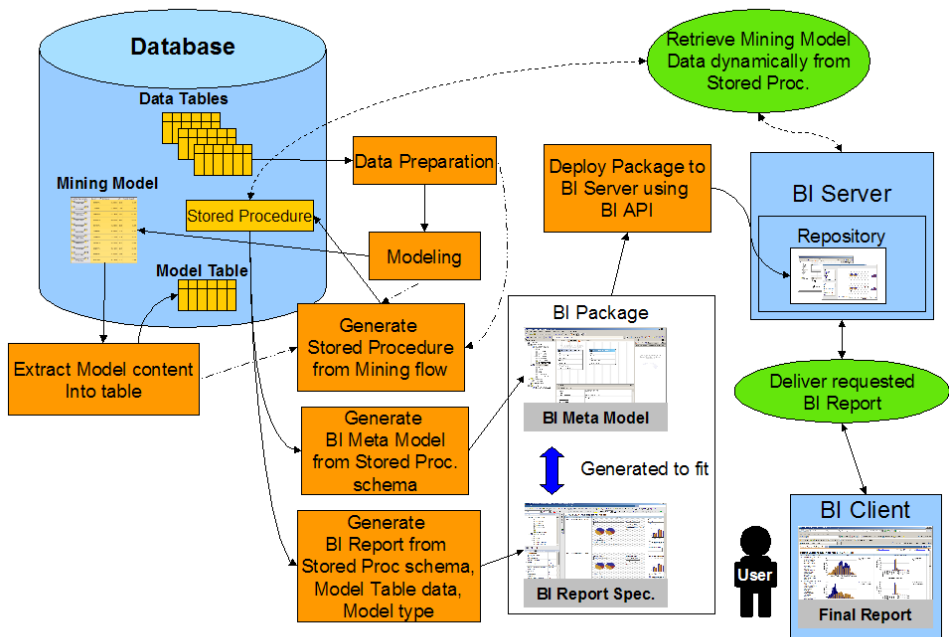


Figure 6: System architecture for deployment of dynamic mining insights

The stored procedure generation is based on the data preparation and mining steps defined by a mining expert (see Figure 6). Often, those steps are defined as data preparation and/or mining flows. The deployment system converts the flow into SQL statements which are placed into the stored procedure. Further, the deployment system incorporates parameters defined by the user. Those parameters are defined as input for the stored procedure and are placed at the proper positions within the SQL statements. E.g. a user could define the maximum number of clusters. As described above, the user simply invokes the report, then the **BI Server** invokes the **Stored Procedure** passing the parameters. The complex flow is completely transparent for the user. The **Stored Procedure** returns data in the same format as the **Model Table**.

5 Implementation

The realized deployment system is based on IBM InfoSphere Warehouse including DB2 with In-Database-Mining. The BI system used is IBM Cognos BI 8. Data preparation and modeling is performed using the Eclipse based InfoSphere Warehouse tool "Design Studio". Design Studio allows defining data and mining flows which can be translated

into SQL and directly executed within DB2. The deployment system is implemented as an Eclipse Plugin for Design Studio. To deploy mining reports to the BI Server the deployment system uses the Web Services API of Cognos BI 8.

The general architecture is based on standardized technologies like relational databases, stored procedures and SQL. Therefore, the deployment system could also be realized for other relational databases with In-Database-Mining like Oracle, Microsoft SQL Server or Teradata. Further, many BI systems provide public APIs to interact with the BI Server like Microstrategy or Business Objects.

6 Summary

Besides the analytical task to generate high quality predictive models the delivery to the right people in time is an important step when turning predictive insight into action. To reach a broad audience the company's BI system is the right vehicle to present this advanced insight in a way the users are used to.

Due to the set up of today's BI environments the process of creating predictive insight and delivering it to the consumers is a complex process requiring several experts in different roles.

The presented system demonstrates how this complex task can be performed in an intelligent, automated fashion to accelerate the delivery process (from hours or days to seconds) and drastically reduce the expert knowledge required.

In this context, not only static mining insight is delivered but the system also allows users to dynamically invoke predictive analytics directly from the BI front-end without the need of deep statistics or mining skills.

As the proposed architecture is based on common standards like relational databases, stored procedures and SQL which are implemented by most vendors, an adoption of the proposed system can be easily realized for most BI environments and product combinations.

References

- [IBM] IBM. Creating Web applications with Miningblox. https://publib.boulder.ibm.com/info-center/db2luw/v9r7/topic/com.ibm.im.blox.doc/miningblox_overview.html.
- [Mic] Microstrategy. Using the Microstrategy Platform to Distribute Data Mining and Predictive Analytics to the masses. <http://bias.csr.unibo.it/lbaldacc/DataMiningWhitePaper.pdf>.
- [PMM] Predictive Model Markup Language. <http://www.dmg.org/>.

Technical Introduction to the IBM Smart Analytics Optimizer for DB2 for System z

Namik Hrle, Oliver Draese

IBM Germany - Research and Development GmbH
Schoenaicherstrasse 220
71032 Boeblingen, Germany
{hrle, draese}@de.ibm.com

The IBM Smart Analytics Optimizer for DB2 for z/OS is a new technology to extend existing data warehouse environments on IBM mainframe systems. It is a workload optimized appliance that enables customers to analyze huge amounts of data in a matter of seconds instead of minutes or hours by delivering unmatched performance. This doesn't only allow “train-of-thought” analysis as interactive scenario but also enables business requests which were simply impossible before. Analytical workloads can now be executed as a online process instead of asynchronous batch processing. A call center employee can for example analyze the customer's behavior pattern while he still is on the phone.

To achieve this new performance, the Smart Analytics Optimizer is implemented as a distributed, In-Memory system where a cluster of computing nodes holds the data in a specialized format in main memory structures. New technology enables the product to perform scans over compressed data without the need of decompression prior to applying predicates. A special partitioning scheme allows the parallel processing of the data with as few locking mechanisms as possible. As the industry trend is showing that an increase of single thread performance is no longer achievable but even standard computers are now delivered with multiple CPU cores, the Smart Analytics Optimizer is designed to exploit this new hardware as good as possible by assigning specific subsets of data to specific cores. The product by itself is running on a cluster where standard instances own hundreds of cores and terabytes of main memory. But even within a single computing core, the product makes use of SIMD instructions to perform parallel evaluation of predicates on multiple tuples.

Besides the raw performance of this new product, the deep integration might even be considered more important. The Smart Analytics Optimizer is not a stand-alone product as it is offered by several other vendors. Instead it extends the existing relational database manager (DB2) by its functionality without requiring any changes to the existing application environments.

Programs, which were connecting to DB2 before just continue to execute their workload against the mainframe database. The internal DB2 functionality then decides when to make use of the new Smart Analytics Optimizer or not. The granularity for these decision is a query block. This implies that a single query with multiple query blocks can be partially executed on the Smart Analytics Optimizer and partially on the mainframe directly. The joined results are returned back to the requesting application by DB2, hiding the complexity of the different execution environments and the required transformations.

Trademarks: IBM logos are trademarks or registered trademarks of the International Business Machines Corporation in the United States, other countries, or both.

Multiple DB2 subsystems can share a single Smart Analytics Optimizer to make optimal use of the attached hardware. Maintenance and configuration is done from DB2 side too. The new product is handled just like any other DB2 resource. Stored procedures and new DB2 commands are used to query the status, perform configuration tasks or control the resources of the accelerator. One of the main goals is to simplify the work of the database administrator. Instead of adding a new resource that needs to be managed, the Smart Analytics Optimizer is an appliance that controls and maintains most of its functionality by itself. The general database administration is now easier because the administrator now longer has to try anticipating all the different query requests against existing tables to create the optimal index structures. By attaching the accelerator, the queries that would fail to match an index, resulting in an expensive table scan, are now routed to the Smart Analytics Optimizer and still answered in a matter of seconds.

Our presentation will give a technical overview about the Smart Analytics Optimizer, typical workloads and the new approaches how these are handled. It explains the new techniques from a hardware and software perspective.

Architecture of a Highly Scalable Data Warehouse Appliance Integrated to Mainframe Database Systems

Knut Stolze¹ Felix Beier^{1,2} Kai-Uwe Sattler²
Sebastian Sprenger¹ Carlos Caballero Grolimund¹ Marco Czech¹

¹IBM Research & Development, Böblingen, Germany

²Ilmenau University of Technology, Germany

Main memory processing and data compression are valuable techniques to address the new challenges of data warehousing regarding scalability, large data volumes, near realtime response times, and the tight connection to OLTP. The IBM Smart Analytics Optimizer (ISAOPT) is a data warehouse appliance that implements a main memory database system for OLAP workloads using a cluster-based architecture. It is tightly integrated with IBM DB2 for z/OS (DB2) to speed up complex queries issued against DB2. In this paper, we focus on autonomic cluster management, high availability, and incremental update mechanisms for data maintenance in ISAOPT.

1 Introduction

For supporting management decisions, more and more companies gather business data in data warehouses (DWH) and evaluate it with the help of modern business intelligence (BI) tools. Reports are used for identifying market trends, conducting risk assessments, performing customer segmentations, and many other analytic tasks. These analyses are considered as business critical by an increasing number of companies and the trend evolves from static analyses towards ad-hoc reporting by a large user group with varying skill levels [Gar07]. The data warehouse systems are therefore faced with new challenges for handling enormous amounts of data with acceptable response times for analytical queries.

Thus, several techniques have been developed for achieving near realtime response times for reporting queries. They reduce the number of required operations for computing results and try keeping processing units fully utilized with reducing the I/O bottleneck when transferring data between different layers in the memory hierarchy. In combination with massive parallel computations, data throughputs can be increased by orders of magnitude.

Special data structures and algorithms have been developed that exploit access characteristics of critical system workloads for reducing the amount of data that has to be processed and efficiently applying typical operators on it. Prominent examples are compressed, read optimized data structures, cache conscious indexes and query operators, as well as operations which are executed directly on compressed data.

Since prices for RAM chips drastically decreased within the last years [Joh02], it nowadays becomes feasible to store the entire warehouse data – or at least the critical part of it – in fast main memory. More and more vendors of modern data warehouse solutions therefore pit on the application of main memory databases (MMDB) or hybrid solutions using a large main memory buffer for the data.

For scalability reasons, typically computer clusters consisting of several nodes are used. Further nodes can be added when the amount of data which has to be handled increases. Moreover, increasing the cluster size may reduce query response times when parallel computations can be efficiently utilized [DG92].

These approaches lead to new challenges in the system architecture which have to be addressed. Because companies rely on the availability and fast response times of their decision support systems, downtimes of latter can become very expensive and have to be avoided. But sometimes errors occur which must be handled internally and transparently, especially in the context of an appliance. Nodes in the cluster might fail and others may take over their parts for keeping the overall system alive. On the other hand, in case the cluster size is increased to scale up the system, the workload should be redistributed for an equal utilization for each of the nodes to minimize overall query response times. Along the same line, mechanisms must be available, which allow a simple upgrade and – equally important – downgrade of the installed software, including automatic forward and backward migration of all internal data structures, ranging from the actual data managed by the system, over the catalog to the actual management of the operating system.

Further problems occur when updates have to be processed on the read optimized DWH systems. They can lead to degenerations of the typically compressed and densely packed data structures. These degenerations have to be removed from time to time for maintaining the system operational and keeping the high performance commitments to user applications. Because the system must be available all the time, offline maintenance operations are not an option. Both – updates to the data and reorganizations of data structures – therefore have to be executed online while concurrent queries are running, without major negative impacts on latter.

All these challenges should be addressed without increasing complexity of system administration. Therefore, an appliance-like approach where most of the maintenance tasks are performed transparently and autonomously without needing interventions by the administrator is the most promising. In this paper we focus on the autonomic cluster management implemented by IBM Smart Analytics Optimizer (ISAOPT) data warehouse appliance [IBM10], which is a cluster-based main memory database management system (MMDBMS).

The remainder of the paper is structured as follows. In section 2 we summarize related work for high availability and administration questions of main memory DWH systems. In section 3 we give an overview on ISAOPT and its integration with IBM DB2 for z/OS. Section 4 discusses the autonomic cluster management features of ISAOPT and how software management (including forward/backward migration) is realized. Data management using incremental updates to synchronize ISAOPT with DB2 is described in section 5. Finally, section 6 concludes the paper and gives an outlook to further research directions.

2 Related Work

ParAccel Analytic Database (PAADB) [Par09] is an analytical high performance DBMS comparable to ISAOPT, implementing a hybrid architecture for both, disk-based query execution with a large main memory buffer and for databases that reside completely in RAM. The data is stored in a cluster with either a shared nothing or shared disk architecture. PAADB uses three types of cluster nodes, all running on commodity hardware. *Leader nodes* provide the interface for communicating with external applications. They are responsible for parsing and scheduling queries, as well as cluster workload management. *Compute nodes* actually store the data highly compressed in a columnar layout and execute tasks scheduled by leader nodes. The main query performance is achieved without tuning mechanisms like MQTs or indexes, but with massive parallel computations. Last, *hot standby nodes* are used for cases other nodes fail to take over their work. Lost data in a failover scenario is recovered automatically depending on the cluster architecture. It is either reloaded from shared disk or from backup replications stored on other nodes.

Vertica [Ver08] also implements a parallel DBMS on a cluster with either a shared nothing or shared disk architecture. High availability is achieved like in PAADB with replicating data. If nodes fail, other ones can take over their workload. The Vertica cluster implements k-safety, i. e. k nodes may fail without causing a system downtime. The overhead for storing replicated data is compensated using aggressive compression schemes on columnar data structures. Since updating these read optimized structures causes some problems, Vertica introduces a write optimized staging area where updates are written to. These changes are applied to the read optimized part by a *tuple mover* during runtime. Several algorithms have been investigated for performing these modifications online in the background to concurrent query execution. [SI09]. Vertica uses a snapshot isolation approach for hiding modifications from concurrent queries without locking overhead.

3 Overview of the IBM Smart Analytics Optimizer

ISAOPT is an appliance consisting of a computer cluster and a parallel MMDBMS. It was developed based on the Blink query processor [RSQ⁺08] for reacting to the trends in modern analytical data warehousing environments. It is aimed to improve warehousing on System z. Because customers using DB2 for z/OS typically pay for computations on mainframes, executing computationally intensive ad hoc reporting or data mining workloads have a noticeable price tag. Thus, ISAOPT is a cost-efficient solution for them, which also delivers a significant performance improvement for such SQL queries.

The main idea is offering an hybrid approach for supporting both OLTP and OLAP workloads with quality of service guarantees close to those for System z. As illustrated in fig 1, OLTP transactions are executed directly by DB2 for z/OS and expensive OLAP queries are offloaded by DB2 transparently to ISAOPT. The appliance is attached to the mainframe via TCP/IP network. DB2 recognizes it as an available resource and offloads query blocks only if it is deemed to be beneficial, based on the decision by DB2's cost-based optimizer.

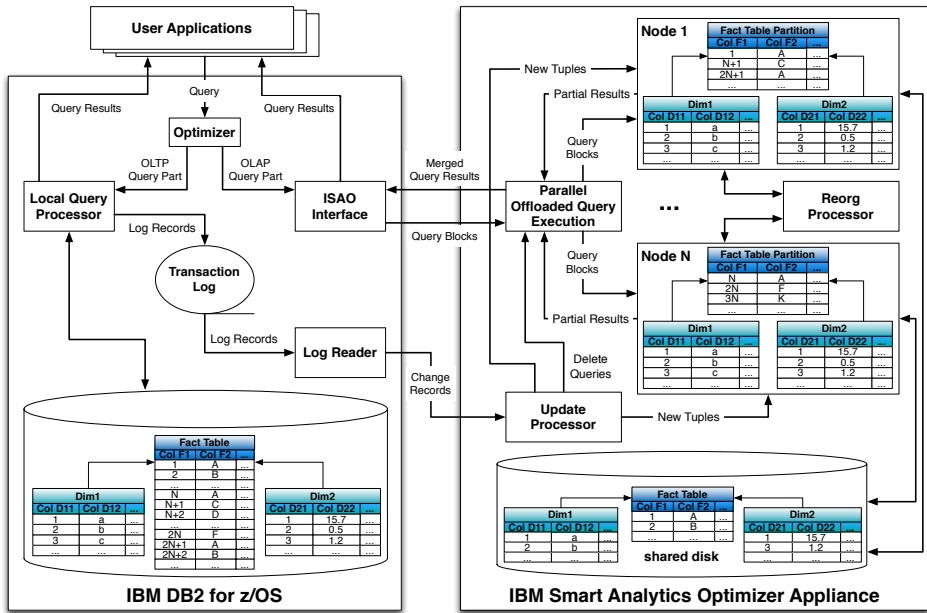


Figure 1: Overview: IBM Smart Analytics Optimizer

Complex analytical queries are split into query blocks which are executed in parallel by ISAOPT on several physical nodes in a cluster. Their partial results are merged and transferred back to DB2 completely transparent to user applications which still use DB2's SQL interface without further configuration or source code modifications. ISAOPT's available main memory and processing power can be scaled by extending the cluster size. Each node uses specialized and dedicated commodity hardware (blades) and therefore executing data warehouse tasks on them is more cost efficient than on general purpose systems like mainframes – with the penalty of lower hardware quality and, thus, a higher chance for outages. But latter is compensated by ISAOPT's cluster monitoring and recovery mechanisms.

Those parts of the warehouse data (called data marts in the following) which are critical for the analytical workload are replicated to ISAOPT. While the data is primarily stored on disks in DB2 (and supported by DB2's buffer pools), it is completely held highly compressed in the main memory of the appliance (and only backed with disk storage for recovery purposes). The used compression technique allows a fast evaluation of equality and range predicates using bitmasks directly on the compressed values and hence avoids an expensive decompression in many cases. [RSQ⁺08] Furthermore, a cache efficient data structure is used which enables the predicate evaluation of queries with vector operations. [JRSS08] Through these techniques, queries can be accelerated by orders of magnitude despite the fact that no workload specific tuning techniques like indexes or materialized views are used. [Dra09] Due to that, no special workload knowledge is necessary by database administrators for guaranteeing high performance of their systems.

4 Cluster Management

ISAOPT uses two different types of nodes. Each query being processed by ISAOPT runs on a single coordinator, which orchestrates the overall query execution, and on all workers. Due to the integration with DB2 for z/OS, the DB2 for z/OS acts as the client sending each query to another coordinator. The decision is based on the current workload of each coordinator, in order to achieve good workload balancing behavior.

Coordinators. A coordinator node does not hold any data in its main memory. It uses its memory to merge partial query results received from worker nodes and to apply post-processing, e.g. sort rows of a result set or compute aggregates like AVG. Since it is not known in advance, how big the intermediate result sets may be, all the coordinators' main memory has to be available – and still it may be necessary to resort to external sort algorithms using external storage, for example.

Another task for coordinator nodes comes with system failures. Since multiple coordinators are available, one of them can take over the data and workload of a failed worker, reducing the impact on other workers – which may already run with high memory pressure. In contrast to PAADB, no hot standby nodes are employed because the hardware resources of such nodes can be put to the task of workload processing without losing any time in case of a system failure.

Workers. A worker node reserves most of its physical main memory to hold the mart data. Only about 30% of the total memory is available for temporary intermediate query results as well as all other operations like monitoring and autonomic system maintenance.

4.1 System Failures

The IBM Smart Analytics Optimizer appliance realizes a shared-disk architecture so that it can easily recover system failures without losing data. All mart data kept in main memory is also stored on disk for backup purposes. The used commodity hardware cannot match the stability of System z, resulting in an increased risk of hardware failures. Reloading the data from DB2 and re-compressing it would be too expensive in terms of time needed for the recovery and CPU overhead for IBM DB2 for z/OS.

Storing the compressed data on a dedicated storage system as-is expedites the recovery process with the least amount of overhead. Mirroring the data in the cluster appears to offer another solution with even faster recovery, but it comes with the penalty that it impacts the most scarce resource in a main-memory database system in terms of scalability: main memory.

4.1.1 Process Failures

A dedicated process is running on each node in the MMDBS cluster. That process may fail, e. g. due to programming errors. Very fast recovery is often achievable with a simple restart of the process and the node is operational again almost instantly. To accomplish that in ISAOPT, all data is loaded into shared memory and will always survive a failing process because it is not owned by its address space. The recovery time is mostly dominated by the time to detect the process failure and to restart it (a few seconds) and not by the time needed for reloading the backup from shared disk (potentially several minutes). Thus, the outage of the overall cluster will be much shorter, delivering on the promise of high availability that is expected by customers today even for unplanned outages.

4.1.2 Node Failures

If a complete node fails, e. g. due to hardware problems encountered at run-time or a problem in the operating system kernel and its drivers, it may happen that the node cannot be restarted (automatically or even manually) and needs to be replaced. In order to keep the cluster operational, redundancy mechanisms are required to detect node failures and initiate recovery steps if necessary. Waiting for the hardware problem to be repaired is not an option in customer environments. The recovery steps include attempts to restart the process on the failing node first, then rebooting the blade, and if that also fails, autonomically removing the node from the cluster and distributing its data and workload to other nodes. All these steps are implemented in the IBM Smart Analytics Optimizer.

The redistribution of the data is based on the shared file system holding the backup information, just created for these purposes. ISAOPT employs GPFS [SH02], a high-performance and fail-safe cluster file system, which has been further enhanced to be fully auto-configurable and auto-recoverable in the context of the ISAOPT appliance. Since GPFS does not yet support IPv6, the requirement came up to automatically configure an IPv4 network for it. The IPv6 automatic configuration features are used to detect which nodes exist in the cluster and to set up an IPv4 class C subnet. That is done during the boot process of each node. A small stand-alone tool is employed for the node detection. DHCP is not being used because its full power was not needed and a DHCP server must not become a single point of failure either. Furthermore, configuring a fail-safe DHCP server automatically would have been a more involved task as well.

4.1.3 System Inconsistencies

In rare cases, it may happen that one of the nodes in the cluster is not running with the correct software version, e. g. because that node was not active in the cluster when a newer (or older) version was installed and activated. Corruptions on the operating system are another problem that the ISAOPT appliance has to be able to deal with automatically. Manual intervention from a support specialist can only be the very last resort. The expectation of users is that such problems are usually detected and recovered by the system itself, i. e. a self-healing approach is required. For that purpose, ISAOPT maintains a so-

called *reference system image*, in which all administrative tasks are applied first. During each node's boot process, the local installation is verified with this reference image and re-synchronized when necessary.

Avoiding such system inconsistencies by using a network boot mechanism may sound attractive, but the network boot server would introduce a single-point of failure. The storage system holding the *reference system image* does not suffer from this problem due to its own redundancy and failure recovery mechanisms.

4.2 Cluster Administration

Cluster administration sums up tasks that are typically performed by a support specialist as a follow-up for hardware failures or newly detected and fixed vulnerabilities. It ranges from replacing broken hardware (e.g. defective DIMMs) over installing new firmware versions and drivers for components like network cards to adding new nodes to an existing cluster for increasing its memory capacity and processing power. Therefore, it may be necessary to remove an existing node from the cluster – while not interrupting the cluster's availability for query processing – or to add a new node to it.

The preparations for such tasks are built on the above described recovery mechanisms, except that the redistribution of the data and workload is explicitly initiated. Due to the shared-disk architecture, other nodes load the data of a node to be removed from the cluster from shared disk and take over the node's workload. The loading of the node's data and workload draining is achieved within a few minutes and can be done in advance so that there is no externally visible outage of the appliance.

Newly added nodes exploit the consistency checks of the operating system level (described above) to make sure the correct version is running. That means, only a very small base system (just 50MB) is needed on new nodes. This base system has to be able to boot the operating system kernel and to perform synchronizations with the reference system image.

4.3 Software Updates & Automatic Migration

A cluster-based MMDBMS must have the capability to switch between different versions of the appliance. That does not only mean to move from the current version of the software to a newer one – it should also be possible to easily rollback to a previous software version. The latter requirement is not often found in today's software products, but it is simply expected by customers of System z, who have the capabilities to upgrade a system like IBM DB2 for z/OS without taking it offline at all if it runs in a cluster. Since ISAOPT is integrated with DB2 like any other service (cf. fig 2), those requirements are transferred from System z directly to it as well.

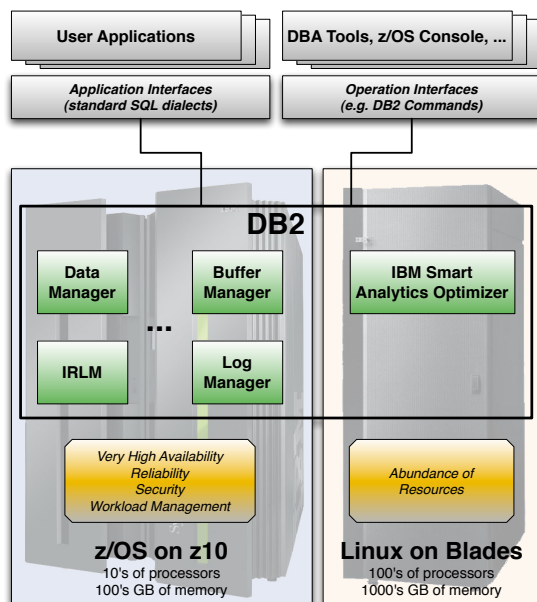


Figure 2: DB2 Integration with zHybrid Architecture

A stored procedure executing in DB2 for z/OS is used to deploy new software versions and to switch between already installed versions in ISAOPT. Forward migration, i. e. migrating structures from an older to a newer version is straight-forward. It can be applied transparently in most cases when the new version is activated for the first time. However, backward migration may be necessary if the newer version may not work as expected (e. g. performance degradations, stability issues, ...).

Transparent migration can not be done when an older version is activated if there were differences in the data structures or the meta data compared to the currently running software version. Therefore, migration functionality is decoupled from the specific version of the ISAOPT server code into its own, light-weight executable. This tool is *always* used in its most-current state. That way, it is ensured that the tool can perform a migration step, even if the current ISAOPT version may not be capable.

5 Incremental Updates and Data Maintenance

Naturally, the data in a data warehouse has to be maintained incrementally. It does not matter if it is a standalone data warehouse system or an accelerator like ISAOPT, which is integrated with DB2 for z/OS and stores its own replica of the original warehouse data. In both cases, reloading a complete mart would just take too long because multiple TBs of data would have to be extracted, transformed, transferred, and loaded again.

Two techniques are being implemented by ISAOPT to update the data and autonomic reorganization is realized as well. Those tasks are running in the background to normal query processing without any downtime to the latter. ISAOPT implements snapshot semantic for all tuples like Vertica (cf. Sect. 2). Unique timestamp IDs (called *epochs*) and an additional range predicate on them are used to guarantee consistent views on the data for running queries while updates are active. [SRSD09] This approach allows a lock-free parallel execution of these concurrent tasks. Furthermore, failing updates can easily be rolled back by undoing the modifications marked with the respective epoch ID.

5.1 Log-based Update

The first update strategy is illustrated in fig. 1. When data stored in source tables in IBM DB2 for z/OS is modified, change records are written into the database log. This delta for committed transactions is read by a log reader, potentially compressed to merge multiple update operations on the same tuples, sent to ISAOPT, and applied there. [SBLC00].

To mark tuples as deleted in ISAOPT, the data of the updated table is scanned in its entirety and the delete epoch for affected tuples is set. Those tuples are not physically deleted and remain in place so that concurrent queries are not interrupted, i. e. setting the delete epoch is transparent for them. New tuples are distributed in the cluster, dictionary encoded, and written directly into the read optimized blocks. A special handling is only required for new values which have no corresponding dictionary code. [SRSD09]

Unfortunately, the prerequisite that updates are logged is not common for customer warehousing environments where many operations usually may bypass the transaction log in order to improve performance, for example when loading a new set of data for sales of the previous day. Nevertheless, if applicable, log-based update is the most efficient method with minimal overhead and can be used for realtime updates.

5.2 Partition-based Update

The second update approach implemented in ISAOPT can be used in all environments where no log records are available or where real time updates are not important and can be processed batch wise in larger time intervals. This solution exploits the fact that big tables are usually horizontally partitioned by ranges where each partition contains a disjoint subset of the table's data. This range partitioning is exploited by ISAOPT to track partitions, which were newly added or have been removed. For removed partitions, the corresponding data is deleted in each replica and all tuples from new partitions are automatically loaded. Additionally, the administrator can specify which table partitions have data modifications e. g. resulting from a LOAD or similar operation. These updates are locally limited to the affected partitions and can be applied with simply re-loading the outdated data.

This solution has a higher overhead than log-based updates because it may happen that some data will be transferred to ISAOPT's replica even if that data actually didn't change. The amount of the overhead depends on the table partitioning granularity, the update frequency on the data warehouse data, and the time intervals when synchronizations between source tables and replicas are initiated.

5.3 Mart Reorganization

When updates are executed, several degenerations can occur which have a negative impact on system performance and scalability. For example, the following degenerations can be observed in ISAOPT:

- New attribute values that could not be considered for dictionary computation in the frequency partitioning algorithm will not have order preserving dictionary codes assigned. They are covered with special cell types as described in [SRSD09] which can not be used for fast range predicate evaluations directly on compressed data. If a tuple has to be stored in the *catch-all cell* no compression can be applied for it.
- Because the frequency partitioning algorithm determines code lengths depending on attribute value histograms when the mart is loaded [RSQ⁺08], shifts in data distributions through updates can have negative impacts on the overall efficiency of the dictionary encoding.
- During updates, tuples must not be physically removed as long as previously started queries are running. With the snapshot semantics, the affected tuples are marked as deleted and leave gaps in the data structures as soon as the last query needing them in its data view finishes. These gaps waste memory and negatively influence query runtimes because the deleted tuples have to be tested for validity in each query snapshot. Since tracking these gaps for overwriting them during following updates is too expensive, they remain in the data structures and require a clean-up.

To address such problems, a reorganization process is needed in cluster-based main memory database systems. Following autonomic principles, a self-scheduled process is used, which is triggered as soon as a reorganization is needed. The decision whether – and which – reorganization should be performed is based on statistics that need to be collected in the system. For example, ISAOPT gathers the number of tuples marked as deleted or stored in the *catch-all cell* and *extension cells* and it also collects automatically information about the executed query workload. Thus, the reorganization task can be a simple removal of cell blocks that only contain deleted tuples, compacting the data by physically removing deleted tuples no longer needed by any query in cell blocks, a rearrangement of the storage layout based on the statistics derived from the query workload, or even the computing and application of a new compression scheme.

Like updates, the internal reorganizations in a main-memory database system like ISAOPT have to be executed online without major impact on query execution. But required system resources like main memory and CPU cycles are critical resources. The reorg process of ISAOPT therefore implements an autonomic adaptation and throttling of resources available for the task at any time, based on the current workload that is active in the system.

6 Conclusion

A highly scalable cluster-based MMDBMS like the IBM Smart Analytics Optimizer has many similarities to traditional disk-based database systems in terms of its architecture. In this paper we presented two important aspects of ISAOPT: cluster management and data maintenance with incremental updates and online data reorganization. Cluster management comprises the automatic detection of system failures and initiating of recovery procedures.

Since ISAOPT is a cluster-based MMDBMS, system consistency across nodes in the cluster has to be ensured, for example in the context of software updates. Mismatches in the software versions are detected and recovery is triggered. The appliance form factor relieves the administrator of all such tasks because they are run autonomically. Thus, administrators only have to take care of deploying software versions and activating the desired version. Migration of data structures and meta data itself is handled transparently by ISAOPT.

Furthermore, no administration efforts are required for maintaining the internal data structures after updates. The appliance autonomically detects degenerations and repairs latter in the background to query processing without major noticeable performance impacts. This guarantees the efficient data structure layout for query processing as well as optimal compression rates to save system resources.

Future directions in the product development will focus on further improving system recovery times and also develop strategies for disaster recovery. Given that the ISAOPT appliance is tightly integrated with DB2, thus it also has to fit into the recovery mechanisms of DB2 itself. Regarding the autonomic management of user data, using ISAOPT's maintenance process for reorganizations as self tuning mechanism is a topic for current research activities. These operations can be exploited to restructure and recompress the internal data structures depending on the monitored system workload with intent to reduce query response times and a more efficient utilization of available system resources.

References

- [DG92] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
- [Dra09] Oliver Draese. IBM Smart Analytics Optimizer Architecture and Overview. Presentation Slides IOD 2009 Session Number TDZ-2711, IBM Corp., October 2009.

- [Gar07] Intelligent Users Use Business Intelligence: From sports teams to traffic police to banks to hospitals, organizations are using analytical tools to turn data into knowledge. www.computerworld.com, 2007.
- [IBM10] IBM Corp. *IBM Smart Analytics Optimizer for DB2 for z/OS V1.1 User's Guide*, November 2010.
- [Joh02] McCallum John. *The Computer Engineering Handbook*, chapter Price-Performance of Computer Technology, pages 4–1 – 4–18. CRC Press, 2002.
- [JRSS08] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *Proc. VLDB Endow.*, 1(1):622–634, 2008.
- [Par09] The ParAccel Analytic Database: A Technical Overview. Whitepaper, ParAccel Inc, 2009.
- [RSQ⁺08] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-Time Query Processing. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 60–69, Washington, DC, USA, 2008. IEEE Computer Society.
- [SBLC00] Kenneth Salem, Kevin Beyer, Bruce Lindsay, and Roberta Cochrane. How to roll a join: asynchronous incremental view maintenance. *SIGMOD Rec.*, 29(2):129–140, 2000.
- [SH02] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.
- [SI09] Gary H. Sockut and Balakrishna R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41(3):1–136, 2009.
- [SRSD09] Knut Stolze, Vijayshankar Raman, Richard Sidle, and O. Draese. Bringing BLINK Closer to the Full Power of SQL. In Johann Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors, *BTW*, volume 144 of *LNI*, pages 157–166. GI, 2009.
- [Ver08] The Vertica Analytic Database Technical Overview White Paper: A DBMS Architecture Optimized for Next-Generation Data Warehousing. Whitepaper, Vertica Systems, 2008.

Interactive Predictive Analytics with Columnar Databases

Martin Oberhofer, Michael Wurst

{martino,mwurst}@ibm.de.com

Abstract: Predictive Analytics is usually seen as highly interactive task. Paradoxically, it is still performed mostly as a batch task. This does not only limit its applicability, it also sets it apart from a task that is conceptually very close to it, namely OLAP analysis. The main reason for considering mining a batch task is the usually very high execution time on large data warehouses. While novel hardware offers the ability of highly distributed execution of predictive analytics algorithms, this level of parallelism cannot be exploited within the traditional row-based database paradigm. Columnar databases offer a solution to this problem, as the underlying datastructures lend themselves very well to parallel execution. This reduces the response time for mining queries several magnitudes for some algorithms.

While making mining faster and more responsive is already nice in itself, the real value of low response times is allowing completely new ways of interacting with huge data warehouses. In this article we give a survey on the opportunities and challenges of interactive, OLAP-like mining and on how columnar databases can support it. We exemplify these ideas on a task that is especially attractive for interactive mining, namely outlier detection in large data warehouses.

1 Introduction

On-line analytical processing (OLAP) enables simplified data analysis exploiting data summarization and aggregation techniques for drilling, pivoting and slicing of the data. OLAP is typically done in a Data Warehouse (DW) environment. Data mining complements the data analysis done through OLAP by automating the discovery of implicit patterns and interesting knowledge hidden in large amounts of data - which is not necessarily stored in a DW and has a broader set of functions such as association rule discovery, classification, prediction or clustering.

While the integration of mining and OLAP as well as interactive mining are highly relevant in many areas, they have received surprisingly little attention so far. Some research on the integration of OLAP and data mining has been done in the 1990's already, coining the term online analytical mining (OLAM). The value of an OLAM environment is typically justified by several reasons. First, a DW environment has information processing tools including tools to profile, cleanse and transform data from various operational sources into the DW. Second, data mining, data needs to have good quality to deliver good results. The effort for cleansing the data for data mining is not needed if data mining is performed in a DW. Third, before a user can effectively mine data, the user might want to do explorative data analysis first. Within an OLAM environment, a user can leverage OLAP

for explorative data analysis and then switch to data mining. In a tightly integrated OLAM environment, switching between the two is seamless for the end user. For the OLAM environment, various authors (e.g. [GRW08] and [HK06]) proposed architectures which work as follows:

Initially, through an Extract-Transform-Load (ETL) process data is extracted from various operational data sources which are typically relational databases. Then the data is profiled, cleansed, transformed to the common model of the DW and loaded. The data model in the DW is usually a snowflake or a star schema in a relational database. Based on the DW model aggregates such as cubes are computed or multi-dimensional databases are built - both to accelerate OLAP query execution but still in relational structures. For the user, there is one integrated UI tool for OLAP and data mining and the user can switch seamlessly between both, e.g. selecting a cube and then apply mining to only this subset. Through this UI the user submits OLAP queries or data mining requests which are received by the OLAM engine which then leverages the Data Mining or the OLAP engine for fulfilling the user request by processing it on data cubes or the multi-dimensional database.

Even though there has been some research and prototypes for a tighter coupling of OLAP and mining, so far the practical use is limited. A major reason for this is the large difference in performance and response times for OLAP queries and for mining procedures. While there are techniques to deliver very short response times for OLAP queries, mining queries often take hours to run. Data mining is still considered a batch job for the following reasons:

- The data volume is typically so large it can not be held in bufferpools which are the main memory area for relational database. Thus, the runtime of data mining processes is typically a batch-style procedure and not an interactive *online* process.
- The cost for large main memory areas for relational databases was too high.
- Existing relational database approaches do not exploit in an optimal manner the new hardware architectures with multi-core CPUs and multi-CPU servers.
- Data compression offered by columnar database techniques is not used in the relational database space yet.

In the following we will show how columnar databases used for accelerating OLAP queries can be used to accelerate data mining as well, enabling both to operate on the same level of interactivity.

2 Concepts of Columnar Databases

Columnar databases are a well-known concept - they date back to the 1970's. Figure 1 shows through an example the conceptual structure of a relational database and in contrast Figure 2 shows the same data in a columnar database. In essence, a columnar databases

Cust ID	First Name	Last Name	Age	City	State	Country
1	John	Smith	31	Santa Cruz	CA	US
2	Alex	Morgan	45	San Jose	CA	US
3	Sarah	Adams	23	Los Angeles	CA	US
4	John	Miller	21	San Diego	CA	US

Figure 1: Structure of a relational database

RID	CUST ID	RID	First Name	RID	Last Name	RID	Age	RID	City	RID	State	RID	Country
1	1	1, 4	John	1	Smith	1	31	1	Santa Cruz	1-4	CA	1-4	US
2	2	2	Alex	2	Morgan	2	45	2	San Jose				
3	3	3	Sarah	3	Adams	3	23	3, 4	San Diego				
4	4			4	Miller	4	21						

Figure 2: Structure of a columnar database

organizes the physical structure in columns rather than in rows. Values stored multiple times in the relational model are stored only once in the columnar model saving space. Using the columnar approach often shrinks the data volume by a factor of 10 to 100. That means a significantly larger amount of data can be loaded into bufferpools in main memory significantly improving query processing performance. Another advantage of the columnar approach is that it is much better suited for OLAP processing where typically only a subset of columns in a table is needed for processing. The columnar database allows access per column whereas the relational model only allows row access even if in each row read only a small number of columns is needed by the query. Thus, the columnar databases approach allows significant reductions on the amount of IO access needed to read the data from the hard disks improving performance. The downside of the columnar approach are insert or update operations which are significantly slower than in a relational model. Thus, columnar databases are very good for OLAP workloads whereas the relational database approach outperforms the columnar database in OLTP scenarios.

Columnar databases store data in a column-centric way. For each combination of column and value there is a list of data points that have this combination (e.g. GENDER=f → r1, r5, r7). These lists are represented in a way that makes intersecting them extremely efficient. This also makes selection operations over conjunctions very efficient, as this operation is reduced to intersecting several lists. Columnar databases are becoming very popular and there are many available products (e.g. Vertica, ParAccel, Infobright) that use different kinds of representation and indexing. In the following we simply assume that there is an efficient mechanism for selecting records based on attribute/value combinations and of intersecting two lists of records.

3 System Evolution Enabling Interactive Outlier Detection

3.1 Columnar Databases and new Hardware Architectures

Columnar databases for OLAP mostly benefit from lower I/O costs through in-memory processing, while they are often far less compute intensive. The opposite is true for data mining. Here I/O can be a minor factor compared to the computation time. Therefore, data mining would usually not directly profit from a columnar in-memory storage. The picture changes, however, if we look at the evolution of hardware:

- The CPU architecture changed from one core to multi-core CPUs. Per core, multiple threads are possible.
- Servers with 8, 16, 32 or 64 multi-core CPUs are normal today providing a previously unparalleled degree of parallelism.
- The prices for main memory significantly dropped and are very affordable now. Thus servers with 2 TB main memory are common computing infrastructure today. A 20 TB large DW in a relational model fits now easily into the 2 TB main memory with the columnar approach.

These capabilities can only be exploited by mining algorithms if the underlying data structures allow for massively parallel execution. This is where the real boost of mining through columnar storage comes from: the ability to solve problems with dozens of threads in parallel, with very fast communication among threads. While there is some existing research work, e.g. [AMS96] and [LOPZ97], research on mining data in columnar databases has again gained increasing attention with the capabilities offered by hardware enabling high degrees of parallel processing. We now show how columnar structures can be exploited for interactive outlier detection.

3.2 A Unified, Columnar Architecture for OLAP and Data Mining

As shown in Figure 3, the new solution architecture has as the two major changes:

- Replacement of the relational database serving as DW with a columnar database
- Replacement of the relational structures for the multi-dimensional database or cubes with a columnar approach

With this proposed solution architecture, in essence we are making the following paradigm shift: The traditional approach in the DW environment was characterized by the fact that only a very limited fraction of the data was in main memory and only a few scans were done in the DB for data mining since they took very long to complete making data mining a batch process. With the new approach, a lot of data is in main memory due to two

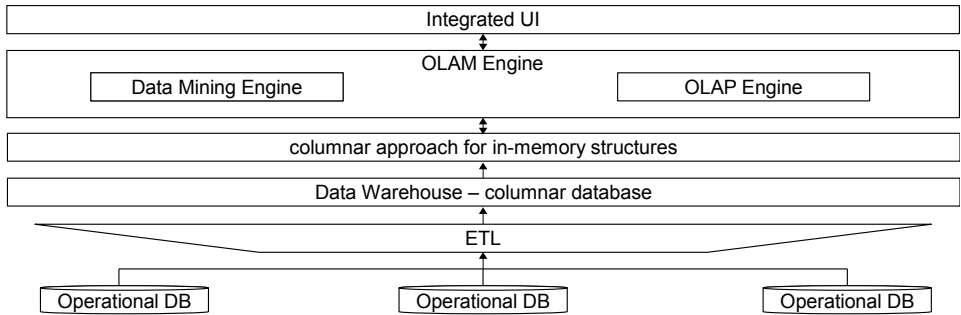


Figure 3: Conceptual overview of new OLAM architecture

reasons: cheap main memory and columnar database approach significantly reducing data volume. With the underlying multi-core, multi-processor server architecture as well as by the fact that columns can be scanned in parallel independently from each other entire new levels of parallel processing for OLAP queries and data mining is possible. In some cases, the scans of data can be completely avoided due to the fact that the data is stored with the columnar approach - examples include min, max or count(distinct) operations.

4 Interactive Outlier Detection in Large Data Warehouses

4.1 The Challenge of Interactive Outlier Detection

Statistical data analysis is one of the key tasks to gain useful insight from large amounts of data. While there are many sophisticated methods for this task (ranging from simple cube analysis to sophisticated predictive analytics models), they have one challenge in common: outliers in the data. Outliers are data points that leaked into the dataset through errors in data acquisition, representation or processing. Typical examples are typos while manually acquiring the data, inconsistencies through false handling of special cases in pre-processing and many others. Unfortunately, such data quality issues appear very often.

Outliers are assumed to behave differently than the majority of data points. However, the opposite does not necessarily hold: just because something appears seldom does not necessarily make it an outlier. It could just be a case that does not occur very often. Depending on the application scenario, this could actually be an especially interesting and valuable case. Therefore, outlier detection is an inherently interactive task of finding unusual patterns in the data, validating and flagging them as outliers or as usual data.

Most algorithms to automatically identify unusual data rely on thresholds. A common approach in statistics, for instance, is to check how far a given value is from the mean value of corresponding data base column. If the age column in a record contains a values that is 100 times the average, then this could be seen as an outlier. However, the difficult problem now is to identify and set the right thresholds. As there is no a priori definition of where unusual but valid data points ends and where outliers starts, the analyst will

subsequently try different thresholds, analyze the list of identified outliers and then re-adjust the threshold accordingly. Also, the analyst will already know that for some cases the predictions of the algorithm can be ignored altogether, as they are known to be usual data points or they are anyway filtered in a subsequent step.

To be able to do an analysis as just described, a system to interactively identify outliers must allow for the following:

1. Combine outlier analysis with slicing and dicing known from OLAP
2. Adjust outlier thresholds interactively
3. Update a list of outliers in near-real time
4. Offer fine grained functionality to filter/ignore some outliers

Existing solutions to outlier detection are mostly based on the idea of finding patterns of usual behavior in the data and then flagging everything that does not match these patterns as an outlier (see [BCV09] for a survey).

4.2 Interactive Outlier Detection on Column-store Warehouses

In the following we assume that our dataset consists of a set of attributes \mathbf{A} in \mathbf{A} , each of which can take a given set of values \mathbf{V}_i (e.g. attribute gender with $\mathbf{V}_{\text{gender}} = \{m, f\}$). We also assume a set of records or data points \mathbf{R} . Each record contains a value for all attributes in \mathbf{A} .

The aim of outlier detection is to identify a subset of records R' of \mathbf{R} that contain unusual combinations in at least one attribute A or in a combination of attributes.

We achieve this by first automatically analysing what the combinations between columns are that are very common. Everything that does not fit into this scheme is considered to be an outlier. This approach has been applied in the past successfully in many different domains (see [BCV09]). We extend this approach in a way that makes it possible to change the settings in a very fine grained way and get the corresponding results in real-time.

Two types of outliers are assumed:

1. Outliers are records that contain values that are statistically very uncommon
2. Outliers are records that contain values that are common - however they represent an statistically uncommon combination

For the first task we simply put a threshold on the frequency of an attribute value pair (or range for numerical fields). Pairs appearing at least in a given fraction of records are considered as normal. All other possible pairs are flagged as outliers. GENDER=f may appear in 40 % of the records, GENDER=fem only in 1%. If the threshold is 5% then the

first record is flagged as normal data point, the second as outlier. Obviously, an optimal threshold could differ for each of the attributes and must usually be set interactively.

For the combination of attributes we refer to the concepts of rules (for an overview see [AS94]): We assume, without loss of generality, that a rule is always written as follows:

$$(\bigwedge A_i \text{ OP } v_{ij}) \Rightarrow A_k \text{ OP } v_{km} \quad (1)$$

where OP can be any of equality for categorical attributes or greater/lesser than for numerical ones. A simple example would be:

$$(AGE < 10) \wedge (BALANCE < 100) \Rightarrow (INCOME < 30) \quad (2)$$

A rule determined from data usually has a confidence score. The confidence score denotes the fraction of records for which the conclusion holds if the premises hold. It can be seen as the strength of a rule. We may assume that the strength of the above rule is 95%. If our threshold for usual behaviour is a rule strength of at least 90%, then we must assume the rule above as usual behaviour and a record with AGE=5, BALANCE=50 but INCOME=100000 would be flagged as outlier. If the threshold would 97%, this would not be the case.

The threshold once the strength of the rule is seen as a reliable describing normal behaviour strongly depends on the application. A suitable threshold must be determined in an interactive process of setting a threshold, determining the outlier this would yield, then resetting it and re-running the outlier detection until a suitable level is found.

It is also possible and quite likely that some columns or rules should be ignored. An example would be a rule that states if a flag is set, another check values is mostly true. We may however know that for a new source system this is no longer required and that violating this rule has no significance. We may also know that the column AGE will anyway not be used in the subsequent analysis, so any violations concerning AGE can be ignored.

It is essential that users can set the confidence threshold and ignore flags on columns and rules in a way that they get immediate feedback on which records would be considered at outliers given the settings, which is not supported by existing approaches. We therefore additionally need a procedure that allows determining in near real-time which records are not covered and must be flagged as outliers. Also, the user should be able to ignore individual outliers, so that the system must be able to keep an *ignore list* of records that should not appear in the result. The overall process is as follows:

1. Build initial rule set and attribute-value counts
2. User (de-)selects columns / rules / records and adjusts thresholds
3. Determine the outliers in real-time
4. Filter deselected outliers
5. Present the outliers to the user (which can then continue with step 2 to further refine results until satisfied)

4.2.1 Identifying Rules and common Attribute-value Pairs

Both common pairs of attribute/value and rules with high confidence can be identified using association rule mining algorithms. As this process only needs to be performed once, efficiency is not critical at this point. Association rule algorithms require a support threshold and a confidence threshold. For the initial run, both are set to a user-defined minimum. This minimum is the lowest possible value for interactive threshold adjustment. All rules and attribute/value combinations are attached with their actual support and confidence for later processing.

4.2.2 Setting Thresholds and Ignore Tags

The user can now adjust the support threshold for attribute/value pairs and the confidence threshold for rules, possibly differently for each column. The user can also tag one or more rules or columns with an *ignore* tag. A rule that is tagged as *ignore* is not evaluated in the subsequent process. A column that is tagged as *ignore* is removed from all rules, which may subsequently lead to the removal of rules with an empty premise (as there is only a single consequence, this case is already covered by the simple attribute/value pairs). Finally, the user can flag records as *ignore* and thus remove them from the results. These settings uniquely identify which records are flagged as outliers, namely exactly the ones, for which either a selected rule does not hold or for which at least one attribute/value pair occurs that is not frequent enough.

4.2.3 Real-time Evaluation of Outlier Records

Having defined the conceptual notion of an outlier, a core question is how to identify the set of outliers in near real-time for interactive work. After a user updated one or several of the settings, a new list of outliers needs to be identified. This is achieved efficiently by exploiting the fact that we are working on a columnar database. Columnar databases store all possible values for a given column and point them to a set of records ids with the corresponding value in that column. They also store all counts, which can be retrieved without accessing the actual records.

For attribute/value pairs, this is very simple, as the frequencies for each attribute/value combination can be read directly from the columnar database. All records that are considered outliers based on this criterion are added to a global outlier list. To capture outliers based on rules, we need a more complex procedure. In a first step, all attribute/value pairs are sorted using an arbitrary criterion. Now, they are stored in a prefix tree. Each node contains exactly one predicate $A_i \text{ OP } v_{ij}$. A path from the root of the tree to a leaf is a conjunction of such predicates. Each node contains a list of all rules, with a premise that is identical to the path from the root to this node. These rules also contain their confidence and the *ignore* flag.

Upon a request, we need to traverse all paths from the root to a leaf node. Please note that this can happen very efficiently in parallel. During traversal, each node is attached with a list of records that fulfill all the predicates that lead to this node from the root. As we

move on from a node to one of his child nodes, we simply intersect the currently attached list of records with the ones entailed by the predicate of the child node (intersection is very cheap). We continue this process until we reached all the leaf nodes. Depending on the implementation of underlying columnar database, the overall process is at most linear in the number of records.

Each of the nodes in the prefix tree is attached with a set of rules. As we traverse a node, we first check which of the rules are activated in the current configuration (based on threshold or ignore tag). All active rules are then evaluated by intersecting their consequence with the list of records at the node. All records not in the intersection are added to a global outlier list. Again, performing this intersection/difference can be achieved extremely efficiently on as the records that contain the consequence can be retrieved directly and intersecting two sets of records is directly supported by the columnar database.

After traversal has finished, the list of records flagged as outliers is further filtered by the list of outliers to ignore using a set difference operation. The result is presented to the user. This list can be sorted by the number of rules violated or any other criterion. Also, a partial list can be shown to the user while the tree is traversed, such that she can see first results while the outlier detection process is still running.

4.3 Evaluation

To evaluate the ability of the above procedure, we used a row-based and a columnar in-memory database, such that I/O would not affect the performance in either case. We used a set of rules, or more importantly literals that would be evaluated. The machine was a UNIX Server running AIX 6 with 8 CPU and 16 GB main memory. We varied the number of records and the number of rule literals and measured the response time on both databases. Figure 4 shows the result.

In both approaches, the response time increases with the number of records and with the rule literals. However, the response time for the columnar storage is much lower than for the row based storage. While even for moderate database sizes, using row-oriented storage leads to response time that are inadequate for interactive work, the column store is still applicable. As all the steps in the rule evaluation can be executed in parallel, a scale-out is rather easy, in cases that the number of records (and thus the response time) exceeds a tolerable level.

5 Summary

OLAP and predictive analytics are two closely related task that can be nicely intergrated with each other. Such an integration is, however, not yet pratically achieved, as the response time for OLAP queries often allows for interactive queries, while the response time of data mining queries is much higher. We showed that highly parallel hardware, in conjunction with the columnar storage paradigm, allows for a massive accerlation of mining

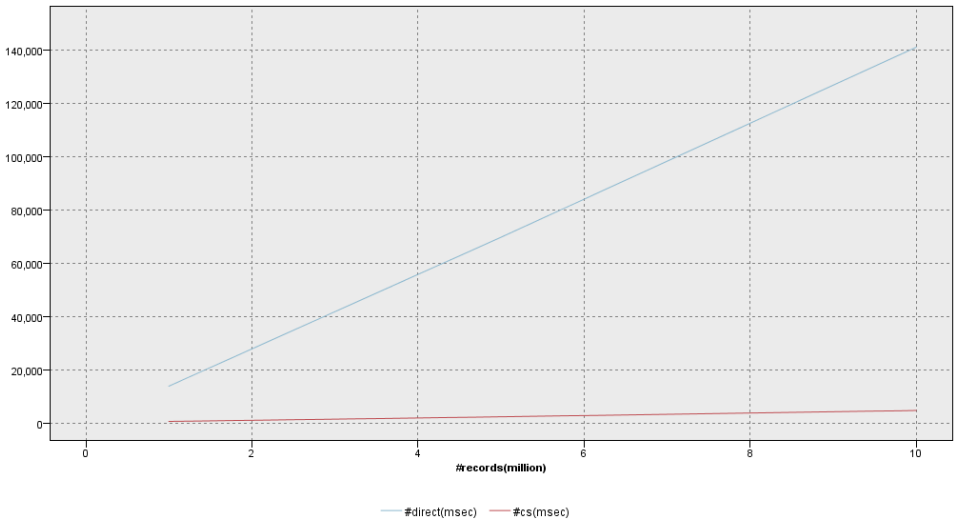


Figure 4: Performance comparison of a relational versus columnar database

queries. This allows for new kinds of applications. We demonstrated this on the task of outlier detection in large data warehouses. We expect to see many further applications of this kind in the near future.

References

- [ABH09] Abadi, D. J. / Boncz, P. A. / Harizopoulos, S.: *Column-oriented database systems*. In: Proceedings of the 35th international Conference on Very Large Data, p. 1664-1665, 2009.
- [AMS96] Agrawal, R. / Mehta, M. / Shafer, J. C.: *Sprint: A scalable parallel classifier for data mining*. In: Proceedings of the 22th international Conference on Very Large Data, p. 544-555, 1996.
- [AS94] Agrawal, R. / Srikant, R.: *Fast Algorithms for Mining Association Rules in Large Databases*. In: Proceedings of the 20th international Conference on Very Large Data, p. 487-499, 1994.
- [BCV09] Banerjee, A. / Chandola, V. / Kumar, V.: *Anomaly Detection: A Survey*. ACM Computing Surveys, Vol. 41(3), Article 15, July 2009.
- [GRW08] Guo, Y. / Rao, W. / Wang, J.: *Research of OLAM Module based on SQL Server*. In: 2008 International Conference on Computer Science and Software Engineering, Volume 4, p. 419-422, 2008.
- [HK06] Han, J. / Kamber, M.: *Data Mining. Concepts and Techniques*. Morgan Kaufmann, 2006.
- [LOPZ97] Li, W. / Ogihara, M. / Parthasarathy, S. / Zaki, M. J.: *New Algorithms for Fast Discovery of Association Rules*. In: 3rd International Conference on Knowledge Discovery and Data Mining, p. 283-286, 1997.

An In-Memory Database System for Multi-Tenant Applications

Franz Färber, Christian Mathis, Daniel Duane Culp, and Wolfram Kleis
{franz.farber|christian.mathis|daniel.duane.culp|w.kleis}@sap.com
SAP AG
Walldorf, Germany

Jan Schaffner
jan.schaffner@hpi.uni-potsdam.de
Hasso-Plattner-Institute
Potsdam, Germany

Abstract: Software as a service (SaaS) has become a viable alternative for small and mid-size companies to outsource IT for a reduction of their total cost of IT ownership. SaaS providers need to be competitive w. r. t. total cost of ownership. Therefore, they typically consolidate multiple customer systems to share resources at all possible levels: hardware, software, data, and personnel. This kind of resource sharing is known as *multitenancy*. In this paper, we will focus on the aspect of multitenancy regarding our new in-memory database system – the SAP in-memory computing engine. In particular, we will motivate the requirements of multitenant applications towards the database engine and we will highlight major design decisions made to support multitenancy natively in the in-memory computing engine.

1 Introduction

Dropping DRAM prices and exponentially growing DRAM volumes changed the way we think about data management. Nowadays, a single blade can hold up to 2 TB of main memory [HPP10]. Database systems running on a cluster with 25 of these blades can store the worlds largest companies' business data, e. g., from enterprise resource planning (ERP) systems, purely in main memory. Why is this observation so important? Because of performance.

The architecture of all of the relevant database systems for enterprise applications is designed to cope with the well-known gap between main memory and external memory (see Table 1). The exchange unit for data transport between external memory and main memory is a *page* containing a number of bytes. To optimize for performance, database-internal data structures (e.g., the B-tree) and query processing algorithms are tailored to paged data access. Furthermore, advanced page-based buffer management and pre-fetching techniques have been developed to shadow the access gap.

Table 1: Access time from main memory and disk []

Action	Time [ns]
Main memory reference	100
Read 1 MB sequentially from main memory	250,000
Disk seek	10,000,000
Read 1 MB sequentially from disk	30,000,000

In-memory database management systems (DBMSs) keep the operational copy of the data entirely in main memory. They only need to access external memory in three particular cases: 1. During system startup to load the main-memory copy, 2. for logging, writing a checkpoint, and recovery, and 3. to persist meta-data and configuration changes. An in-memory DBMS can choose to rely on paged data handling (e.g., paged buffering) for these tasks. All other operations run purely in main memory. For them, the in-memory DBMSs can avoid paged data handling and encode data in contiguous memory DRAM areas (i.e., arrays). Storage structures and query processing algorithms can be tailored to work well with arrays instead of pages. This distinguishes them from a classical disk-based system equipped with a large buffer space. Such a system is entirely page-oriented and, compared to in-memory DBMSs, suffers from the organizational overhead imposed by the page-centric data handling [GMS92].

Besides RAM developments, the multi-core trend also has a substantial impact on data processing systems. Because the “frequency rally” stopped in 2006, software vendors like SAP now cannot rely on frequency-based performance speed-up anymore. Instead, we have to find ways to parallelize our software to – ideally – scale linearly with the number of available cores. Current hardware architectures have up to 64 cores (on eight sockets) per blade. All 64 cores have shared access to the main memory. We again mention that these blades can be switched together in a cluster, resulting in a highly parallel system with local shared-memory access [HPP10].

The *SAP in-memory computing engine* (IMCE) is an in-memory DBMS designed for a multi-blade, multi-core hardware architecture. It is a relational database system with SQL and full ACID support. Apart from providing standard database functionalities, the IMCE is also aimed at a sound integration with SAP business applications. Certain requirements of enterprise applications have influenced the design and functionality of our in-memory database system. A particular requirement, which we like to showcase in this paper, is *multitenancy* – the ability to handle multiple clients or *tenants* within the IMCE.

The remainder of this paper is organized as follows. In the next section, we discuss the concept of multitenancy. Then, in Section 3, we give an overview over the key features of the SAP in-memory computing engine. Section 4 lists the requirements posed by multitenant applications towards the data-management layer. Section 5 highlights the implications of multitenancy support in our new database system, before Section 6 concludes the paper.

2 Multitenancy

The classic *on-premise* software distribution model for enterprise software (e. g., ERP systems) involves software licensing by a customer, software installation on his hardware, and software maintenance by his personnel. Especially smaller and mid-size companies can easily get overburdened or even distracted from their core competencies by the task to run their own IT. Although running enterprise software is mission critical to these companies, the need for cost and complexity reduction makes outsourcing IT by obtaining *Software as a Service* (SaaS) viable.

SaaS solutions are completely hosted by a service provider, i. e., the software and the data resides on the provider's data centers. Thus, service providers operate the companies' IT system off-premise. Compared to on-premise solutions, they can thereby alleviate companies from the risk of over-provisioning/under-provisioning and under-utilization/saturation by "elastically" adding and removing resources. Companies, on the other side, only pay for the service, either by subscription or on a pay-per-use basis.

The reason why all this works for the SaaS provider is *resource sharing* among customers at all levels to reduce total cost of ownership (TCO): hardware, software, data, and personnel. In a SaaS system, a customer is called *tenant*. Several tenants share the hardware of a machine, where the mapping between a tenant and its hardware is kept flexible. If a tenant grows, e. g., because of growing data volumes or because of a growing user base, the service provider can decide to migrate it on a larger machine (*scale up*) or to dedicate more machines to the tenant (*scale out*). Also, if hardware capabilities grow, the service provider can decide to put more tenants on a single machine.

Tenants can also share the same software. This is especially true for standard enterprise software as developed by SAP, but also for generic components, like database systems, application servers, and repositories. Furthermore, because the provider controls both, hardware and software, he is not obliged to port the software to all commercially relevant hardware platforms. Rather, he is able to specially optimize for the hardware of his choice.

Applications running on behalf of multiple tenants can also share (general or public) data, for example country-specific information like population, currency, exchange rate, and gross national product (GNP), e. g., for analytics. Furthermore, multitenant applications can share meta data. Finally, the personnel operating the data centers can share a single administration framework to unify software and hardware administration for all tenants.

As already stated, SaaS is not only about software. Along with the software, the data goes to the SaaS provider as well. The question is, how do SaaS providers manage data from multi-tenant applications. Of course, the solution lies in the realm of database systems. We see three alternatives how to manage data from multiple tenants with the help of database systems [JA07]:

- *Shared machine*: The DBMSs running on behalf several tenants share the resources of a single machine. The advantage of this approach is a good degree of isolation between tenants and simple tenant migration. Especially when the DBMS is installed in a virtualized environment (e. g., XEN, VMware, Amazon EC2), DBMS migration

is as simple as virtual machine migration. Furthermore, standard database systems can be used to realize the shared machine approach; no tenant-specific functionality is required. Due to the maximum degree of isolation, every tenant can extend or modify its database schema without restrictions (which is important for tenant-specific customizations). When a DBMS instance crashes, no other tenants are affected. On the flip side, the various DBMSs installed on a single machine compete for resources, like memory pools, communication sockets, or execution threads. Managing these resources generates overhead in each running system. Additionally, there is no centralized control to govern shared resources (except for the operating system, which is, however, not aware of the specific resource consumption patterns of multitenant database systems). Finally, administering one DBMSs instance per tenant without additional (external) tools can become a cumbersome task when a large number of tenants has to be supported.

- *Shared table*: All tenants share the same DBMS instance, the same schema, and store their data in the same tables. To distinguish tenants from each other, a special column containing a tenant identifier is inserted in each table. Again, standard database systems can be used to implement this approach. Compared to shared machine, system resources are shared/controlled, and maintenance applies to only one instance. However, tenant migration becomes more complex, because tables and disk volumes are not isolated. In the case of a crash, all tenants are affected. Furthermore, database queries have to be rewritten to restrict results to the queried tenant. Finally, tenant-specific extensions always influence other tenants.
- *Shared DBMS instance*: All tenants share the same DBMS instance. To isolate the tenants, each tenant runs in a separate process, has its own tables, and its own external-memory volumes. However, resources between tenants, e. g., memory pools, communication sockets, and execution threads can be shared (by a pooling infrastructure that is common for all tenants). Sharing of meta data, i. e., the database schema, is also possible. Clearly, this approach requires that the DBMS is made aware of multiple tenants. However, it combines the best of both previous approaches: Tenants are isolated, maintenance applies to one instance only, resources are shared/controlled, migration can be implemented based on tenant-specific disk volumes, (meta-)data can be shared, a crashing tenant process does not tear down other tenants, and tenant-specific extensions do not influence other tenants.

Because the third approach seems most reasonable, the SAP in-memory computing engine supports multitenancy by sharing the DBMS instance. We will return to the details in Section 5. First, we would like to give an overview over the features of our in-memory database system.

3 The SAP In-Memory Computing Engine

The SAP in-memory computing engine is a relational database system that keeps the primary data copy in main memory. The target hardware consists of a cluster of blades, where

each blade is typically equipped with up to 64 cores per blade and up to 2 TB main memory (at the time of writing). Of course, to avoid data loss during power failures, the system needs a persistence on external storage as well as logging and recovery mechanisms (see Section 3.8). However, the principal idea is to optimize the database system towards main-memory access and keep the access frequency to external memory as low as possible. To an end user, the system provides the usual functionality one might expect from a database system: it supports the relational model, has a SQL interface, backup and recovery, and full ACID support. In the following, we will discuss some key concepts of the in-memory computing engine.

3.1 Row Storage and Column Storage

Tables in the in-memory computing engine can be stored column-wise or row-wise. The storage mode can be defined by the user. Both storage types have their advantages regarding access behavior. The column store supports set-based read-intensive data processing along columns, for example, aggregation queries in online analytical processing (OLAP). The row store naturally supports row-based and update-intensive operations, like single key lookup or single row insertions. It serves best performance in applications with online transaction processing (OLTP) characteristics. The storage mode is transparent to the query engine, i. e., queries can freely combine data stored in both table formats. Tables can also be converted on the fly. We describe the internal storage layouts for row and column store below.

3.2 Compressed Column-Store Layout

To improve memory bandwidth utilization and to keep the memory footprint of enterprise data storage small, the column-store data in the IMCE is compressed. Especially, lightweight compression schemes [BHF09, LSF09] are applied, where the CPU overhead for compression and decompression does not overshadow the gain of reduced memory bandwidth utilization. Furthermore, data stored in columns is particularly well-suited to compression, because all values of a column stem from the same value domain.

All columns in the column store are compressed using dictionary coding: The values of the column are replaced by integers (value IDs) that point to the original value, which is stored in a separate sorted array. For example, assume we have a column where each entry is one of four colors: [red, green, red, blue, white, red, ...]. The dictionary-encoded variant consists of a sorted dictionary [blue, green, red, white] and an array named *column vector* [2, 1, 2, 0, 3, 2, ...] containing the dictionary positions. We assume that the size of the dictionary (i. e., the distinct-value cardinality of the column) is known in advance. Therefore, we can encode each integer with the minimum number of bits (two in our example) and pack them in a contiguous memory location. In our example, this contiguous location would contain the following binary string: 100110001110... If string values are

stored in the dictionary, the dictionary can be compressed using prefix compression. On the compressed column vector, we apply more lightweight compression schemes, such as run-length encoding or cluster coding [LSF09].

3.3 The Delta Store

The above described dictionary encoding does not allow cheap modifications or insertions. For example, if a new column value appears due to an insert operation, it has to be placed at the correct position in the sorted dictionary. The positions of the following entries have to be shifted, requiring an update to all the affected integers in the column vector.

To solve this problem, the column store provides a write-optimized storage location called *delta store* (the read-optimized store is called *main store*). The delta store is also column-oriented and utilizes dictionary coding. In contrast to the main store, the dictionary is not sorted. For fast access, a CSB+ tree index [RR00] is generated on the dictionary, which maps the value ID (as index key) to the position in the unsorted dictionary (as index value).

The delta store requires more space than the main store. Due to the unsorted dictionary and the additional CSB+-tree lookups, the performance is also slightly worse than the main store. Therefore, from time to time, the delta store is merged together with the main store [KGT⁺10]. To keep the system accepting updates during a merge, a second delta store is created for each column. The secondary delta is declared to be the primary delta after the merge has finished. Furthermore, while merging a table, the merged table is written to a new main store. The old main store can then still be used to process read queries during the merge phase.

3.4 Row-Store Layout

The row store keeps rows intact and organizes them 16 Kb pages which are kept in a linked list, one per table. Variable-length fields are stored in “referenced mode”, i. e., the data pages contain a pointer to the memory location, where the variable length value is stored. Because of fixed-length rows, each row can be identified by a its physical address. This *row ID* consists of a page identifier and an offset. In contrast to the column store, the row store is not compressed. This allows simple and fast insertion: When a new row has to be inserted, the *page manager* searches for a free slot in an existing page. If no such page exists, a new batch of pages (64 MB) is allocated and the record is placed in a slot of a new page. To speed up key-based lookup operations, columns in the row store can be indexed using a cache-aware index structure. Such an index maps a value to the row IDs of its occurrence. Indexes exist only in main memory and are built on the fly when the table is loaded from external memory.

3.5 Insert-Only and Multi-Version Concurrency Control

Many business applications have the need to keep historical data, e. g., for legal purposes. Therefore, they do not want to update and overwrite data in place. To support these applications, the column store of the in-memory computing engine follows the *insert-only* approach, i. e., physically, data is never updated in place. Rather, when a row is updated, the new values are inserted into the delta store and the old values from the main store are marked as overwritten. Likewise, deletions mark a row to be deleted but do not physically remove the entry. In the IMCE, the so-called *consistent-view manager* (CV manager) keeps track of these changes. As an analysis of several customer systems has revealed that many enterprise applications are not update intensive [Pla09]. However, from time to time, the in-memory store should be freed from “old” entries. These old entries can be removed during the delta merge. To fulfill the requirement that the data can still be queried, the merge process places old rows into yet another store, namely the *history store*, which can be placed on external memory to reduce main-memory consumption.

Insert-only is not a prerequisite for optimistic transaction synchronization [HR01]. However, the two concepts go well together, because concurrently running transactions have to keep their write sets in optimistic concurrency control (OCC) anyway. The IMCE combines the insert-only approach with multi-version concurrency control (MVCC). Because insert-only does not physically remove or overwrite rows, the CV manager can keep track of multiple versions. The CV manager can also make sure that every read transaction can operate on a stable snapshot of the database at the time of transaction start (snapshot isolation). Read/write and write/read conflicts between concurrent transactions can therefore be avoided, thus facilitating transaction parallelism. To resolve write/write conflicts, we do however not rely on an optimistic concurrency control scheme. Instead, we use classic row-level locking. This avoids unnecessary transaction rollbacks due to overlapping write sets. The IMCE provides transaction levels read committed and repeatable read. We omit the discussion of insert-only and MVCC concepts in the row store for brevity.

3.6 Data Distribution

The SAP in-memory computing engine runs on a cluster of blades, where the blades are interconnected by ethernet. The persistent store resides in a network-attached storage (NAS) or a storage area network (SAN), connected to all blades [MSLR09]. Because each blade has access to the same database, we would classically call this architecture *shared disk* [Sto86]. However, because the *primary* copy of the data does not reside on disk, but in the main memory private to each blade, we characterize it as a *shared-nothing* architecture. With each blade having access to the same disk, the system has the chance to survive blade failures: The data stored in the main memory of a corrupted blade can be recovered from disk and re-loaded to the other blades automatically. Alternatively, a backup blade can take over. We will come back to implementation details of distribution in Section 5.1.

3.7 Parallel Data Processing

The in-memory computing engine allows to horizontally partition tables and distribute the partitions across the blades. Several partitioning criteria are possible: round robin, range-based, hash-based, etc. Running queries against partitioned data naturally can be executed in parallel (data parallelism). Distributed queries are implemented on the basis of a distributed query plan, distributed query processing algorithms, and a distributed plan execution engine. To make full use of all compute resources, within one blade, the system supports most kinds of query parallelism: *inter-query*, *inter-operator*, and *intra-operator* parallelism [Rah94]. For fast scan and aggregation, special *single instruction multiple data* (SIMD) instructions are applied, that can, for example, sum up four integers in parallel [WPB⁺09]. The IMCE does not support the concept of *pipeline parallelism*. Rather, every operation in the query plan runs to completion, before the result is sent to the next operation. This allows us to optimize the data structures keeping intermediate results and to save communication costs, when network access is required to ship intermediate results.

3.8 Persistence, Logging, and Recovery

Just like a disk-based DBMS, an in-memory DBMS has to ensure that after a power failure the database can be recovered. The SAP in-memory computing engine is built on a page-based persistence, i. e., a log entry can be written for a physical page (physical logging) [GR93]. The log protocol adheres to write-ahead logging (WAL), i. e., before a transaction commits, the log buffer is written to disk. To avoid undo logging, the buffer manager implements the shadow-page concept [Lor77]. Due to the WAL protocol, pages can be written to external memory in deferred mode (after transactions have committed).

The column store implements logical logging to keep track of the modifications on the delta store. The logical log is written to a *virtual file*, which consists of pages provided by the page-based persistence layer. Thereby, the logical log is recoverable. Note, the delta store itself is not written to external memory. Rather, it is built from the logical log during system startup. For the row store, as special differential logging technique is applied, which enables parallel log streams to multiple external disk drives [LKC01]. All data and log information is written into *disk volumes* – files provided by the operating system.

4 Data Management Requirements of Multi-Tenant Applications

The in-memory computing engine allows multiple tenants to share one instance, i. e., it follows the *shared DBMS instance* approach described in Section 5.2. Instance sharing is implemented by adding tenant-management logic in the database system. We will see how this works in the next section. First, let us look at the data management requirements posed by multi-tenant applications, such as Business ByDesign [ByD] – SAP’s enterprise solution for small and mid-size companies.

In the following, we assume a classical three-tier software architecture, where the *presentation layer* is separated from the application logic running on various application servers, and the *application layer* is separated from the *database layer*. Orthogonal to this software stack, a multi-tenant application needs software to maintain the stack. We omit the discussion on the aspects of multitenancy to the presentation layer (e. g., tenant-specific customizations) and the application layer (e. g., tenant-specific application logic) and focus directly on the requirements posed on the database layer and the database-specifics of the maintenance software.

For an application, the overall requirement to the database system is *transparency of resource sharing*. The application should operate on a tenant database, as if each tenant would possess its own database instance. For example, the application should be able to open a database connection for a specific tenant. All queries via this connection are then local to the connection's tenant. This way, the application does not have to deal with "special logic" to query tenant-specific data (such as having to inject a tenant ID to all SQL queries). The application should also be disallowed to directly access data from other tenants (if an application requires interaction with other tenants, it has to use the standard communication channels, e. g., via middleware software, such as SAP Process Integration [PI]). Transparency of resource sharing also means that a tenant should not be influenced by other tenants whom resources are shared with. For example, when a tenant crashes, it should not tear down other tenants. Furthermore, when a tenant requires peak database performance, other tenants sharing the same resources should still meet their service level agreements (SLAs).

In the requirements discussion so far, we have neglected the fact that the application running on a multi-tenant database system also has to support multiple tenants and needs to share resources. The database-specific fraction of these shared resources are schema artifacts and shared table data. To distinguish them, we classify tables as: 1. *tenant-independent tables*, 2. *tenant-dependent tables*, and 3. *tenant-private tables*. Tenant-independent tables contain information useful for any tenant, for example, currency exchange rates. Tenants are not allowed to modify these tables. Because to the application, every tenant has the same initial schema, the database should support to specify an initial schema (called *tenant template* in the following) from which new tenants can be created by cloning. The tenant template contains tenant-independent tables and tenant-dependent tables. Initially, both can contain data, for example, default values (but do not have to). A running tenant stores its data in the tenant-dependent tables. Applications should be allowed to extend the initial schema with tenant-private tables. It should also be possible, to add new columns to existing tenant-dependent tables. This is important for application customizing.

Of course, the maintenance software of a multi-tenant application cannot be transparent w. r. t. resource sharing in the database layer. It has to organize resource sharing. The database system should provide services for tenant life-cycle management, such as creation (see above) and dynamic tenant startup and shutdown. It should make tenant relocation, backup, and cloning possible without downtime or significant performance degradations. Relocation is required when tenants have to be moved to faster/larger machines or if they want to go on-premise. Closing is often required to create test tenants.

The database system should also allow to define tenant-specific quotas, e.g., to model SLAs. Furthermore, to support flexible software roll-out scenarios, the database system should provide for tenant-specific release management. Of course, this only applies to the database schema and tables, not to the code base of the database system itself (which provide tenant-specific release management without tenant relocation). To react on customer-specific failures and performance glitches, tenant debugging, tracing, and monitoring facilities are required. Finally, the database system should allow to replicate tenants for high availability and load balancing. Please consult [AJPK09] for further readings.

5 Tenant Integration in the In-Memory Computing Engine

The in-memory computing engine implements multitenancy through the shared DBMS instance approach. The basic concept to achieve transparency of resource sharing is *tenant separation*. The IMCE carefully distinguishes between artifacts that have to be separated between tenants and those that have to be shared among them. We will come back to this point in Section 5.2. Because support for multitenancy is intermingled with the distributed system aspects of the IMCE, we turn our focus first to distribution.

5.1 Distributed Instances

The topic map in Figure 1 shows the elements of a distributed IMCE instance and how they are related. A distributed instance consists of multiple database servers, each of which runs in a separate operating system process and has its own disk volumes. The database servers of a distributed system may be distributed across multiple hosts – nevertheless it is also possible to run multiple database servers on one host. The key principle of executing database operations is to run the operations at the location, where the data resides. Therefore, database servers may need to forward the execution of some operations to other servers that own some data involved in the operation. In a data distribution scenario, the database clients do not need to know about the distribution. They may send their requests to any database server. If the server does not own all data involved, it will delegate the execution of some operations to other servers, collect the result, and return it to the database client (performance considerations nevertheless make it desirable to send the request immediately to the processing node).

In a distributed IMCE system, a central component is required that knows the topology of the system and how data is distributed. In our system, this component is called *name server*. The name server holds for example the information, which tables of table partitions are located on which IMCE database server. In a multi-tenant system, the name server is hierarchical in a way, that the global nameserver knows the assignment of tenants to IMCE database servers, whereas the tenant-specific nameservers hold the information about table locations.

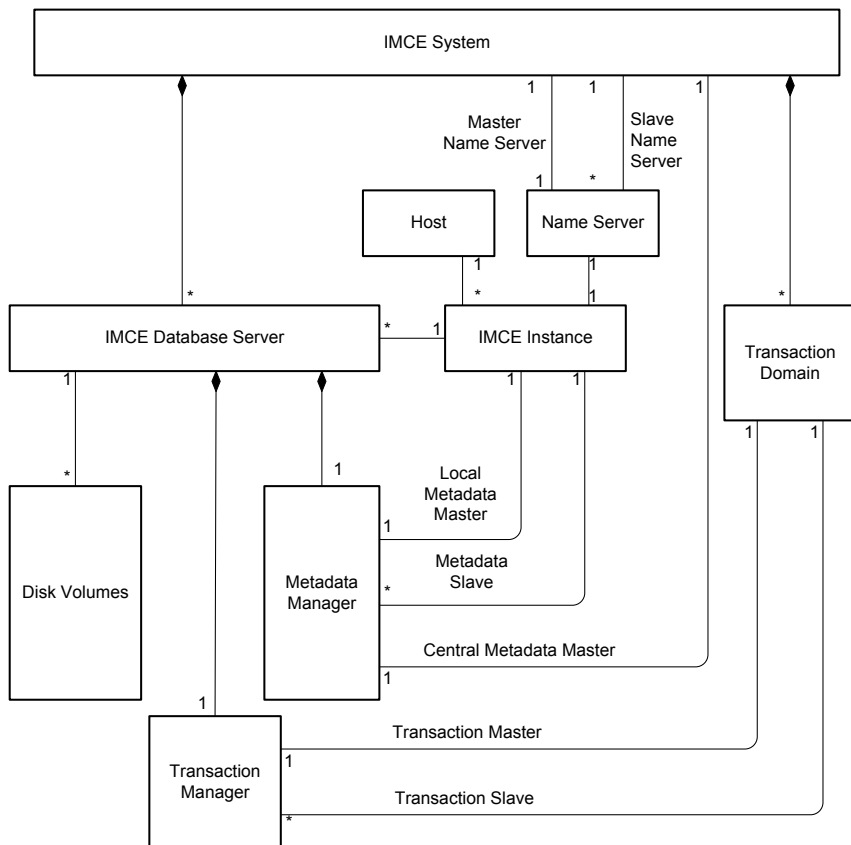


Figure 1: Structure of Distributed IMCE systems

When processing a query, the database servers ask the name server about the locations of the involved tables. To prevent this from having a negative impact on performance, the topology and distribution information is replicated and cached on every host. In each IMCE system, there is one master name server that owns the topology and distribution information. This data is replicated to slave name servers that run on each host. The slave name servers store the replicated data to a cache in shared memory from where the IMCE database servers on the same host can read it (if belonging to the same system).

In a data distribution scenario, the partitioning can be done table-wise or by splitting tables horizontally. With table-wise partitioning, the name server assigns new tables to a IMCE database server based on the current distribution of tables and load (number of tables assigned and resource consumption of the database servers). Data for this table will reside only on that database server, if no additional replicas are specified. It is also possible to specify that a table is split over multiple IMCE database servers. Initial table partitioning is done by the name server based on a size estimation specified by the application. When records are inserted into a partitioned table, they may be distributed to other IMCE

database servers based on the table's partitioning specification. In a multi-tenant IMCE system, partitioning is done tenant-wise.

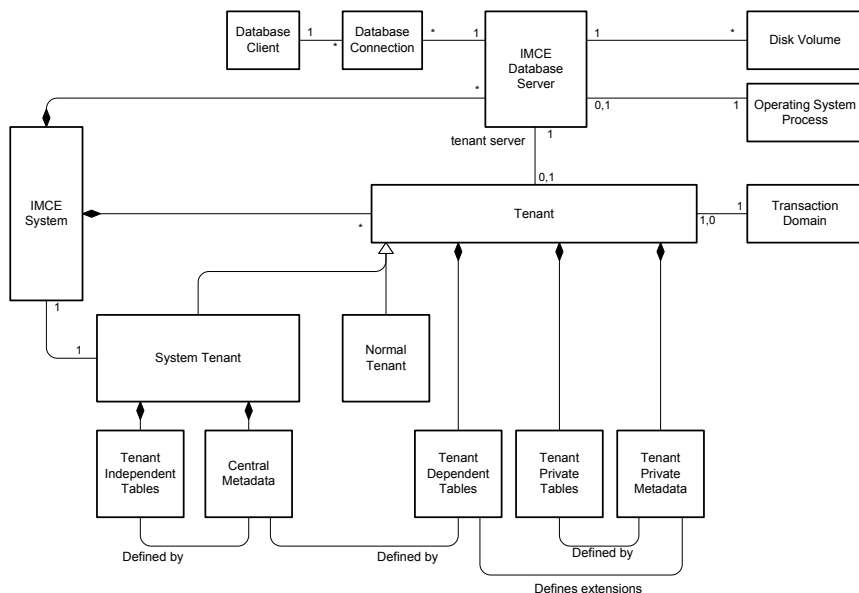
The master name server of an IMCE system is a critical central component in a distributed setup. To ensure high availability, it is possible to have additional name servers as backup master name servers (standard configuration is two). During normal operation, the backup master name servers receive all replicated data like the slave names servers. If the master name server fails, the backup server negotiate and one of them takes over the role of the master server.

Analogous to the topology information and nameserver, each IMCE database server contains a meta-data manager that provides meta-data-related interfaces to all other IMCE components. Meta data are for example table definitions or types of the columns. In a distributed IMCE system, meta data is defined and stored centrally and replicated to all database servers. One of the database servers takes the role of central *meta-data master*, of which there is one per IMCE system. Only on the host running the central meta-data master, global meta data can be created.

On each host, there is a *local meta data master* that receives replicated meta data from the central master. The local meta-data master makes the replicated meta data available to the IMCE database servers on the same host using shared memory. These database servers are called *meta-data slaves*. Meta-data slaves have only read access to the replicated central meta data. Central meta-data master and local meta-data master are not separate server processes, but are hosted by specific IMCE database servers. Meta-data replication is handled transparently by the meta-data managers. All other IMCE components use the meta-data manager interface. Therefore, the replication of meta data is completely hidden from them. For read access, meta-data replication is also transparent for database clients. No matter to which server a database client is connected, it can read all central meta data.

Because central meta data is created in the meta-data master only, database clients that need to create or change central meta data need to connect to the meta-data master. A second type of meta data, *local meta data*, can be created on the local database server. This type of meta data is local to the database server where it is created and is not shared with others, even on the same host. This feature is used in multi-tenant systems for defining meta data that is private to the tenant (see Section 5.2).

To ensure transactional consistency in distributed setups, the IMCE supports distributed transactions. Each IMCE system can have multiple *transaction domains* to which the IMCE database servers are uniquely assigned (as shown in Figure 1). Distributed transactions may span only IMCE database servers within the same transaction domain. In a transaction domain, there is one IMCE database server that has the role of the transaction master, while the others act as transaction slaves. The transaction master also coordinates the two-phase commit protocol.



5.2 Multitenancy

Since SAP R/2, multitenancy is to some degree achieved with the *client concept*: All client-dependent tables contain a client column that is used by the application server to distinguish multiple clients (shared table approach). The client column can be used for partitioning and clustering at the database level, but for conventional DBMSs, the client identifier is just another column, i. e., conventional DBMSs are not aware of the client concept. In IMCE, this is different, because it offers support for multiple tenants already at the DBMS level (shared DBMS-instance approach).

As we will see, the tenants of one IMCE system share common meta data, but for the actual tables that hold application data, there are separate instances per tenant. This means that each tenant has its own set of application-specific tables storing all the tenant-specific data. As a result, a tenant-identifier column is not needed in the tenant-specific instances of the table. The tenant identifier would always contain a constant value.

The topic map in Figure 2 explains, how important concepts relevant for multitenancy are related in the in-memory computing engine. An IMCE system with multiple tenants is a distributed IMCE system, where each database server uniquely belongs to one tenant. Tenants run in different operating system processes with their own virtual memory. Additionally, they also have their own disk volumes. This separation allows us to isolate tenant crashes or failures and to recover from them tenant-wise. In the following, the term *tenant server* is used to refer to the database server that belongs to a specific tenant.

Table 2: Tenant Separation and Types of Tables

Type	Table Content (Data)	Meta Data
Tenant-Independent	Stored in system tenant; Read access from other tenants	Stored in system tenant; Read access from other tenants
Tenant-Dependent	Independent instances of the table exist in each tenant with content that is private to the tenant	Table definition is stored centrally in system tenant with read access from other tenants; tenant-specific extensions (additional columns, etc.) are stored as private meta data
Tenant-Private	Stored as private data of the tenant with no access from other tenants	Stored as private meta data of the tenant

To support data and meta-data sharing in multitenant applications (such as Business By-Design), the in-memory computing engine provides the three table categories introduced in Section 4: tenant-independent tables, tenant-dependent tables and tenant-private tables. To implement these table types, each IMCE system has one special tenant called the *system tenant* (other tenants are called *normal tenants*). The system tenant contains tenant-independent meta data and application data, as well as the tenant-dependent meta data. Both can be accessed in read mode by normal tenants. Tenant-independent data are, for example, country names or time zones for currency rates. Tenant-independent data can be relevant to all tenants and is delivered by SAP as built-in content. As the owner of the global meta data in a distributed IMCE system (see Section 5.1), the system tenant also manages the central meta data of tenant-independent and tenant-dependent tables which is, like the built-in content, available to all other tenants for read access.

Normal tenants can define their own private meta data (i. e., new tables, columns, views, functions), which is only visible to them. Furthermore, The data from normal tenants itself is also isolated: In the context of one tenant, data from other normal tenants cannot be accessed directly. If a client needs access to more than one normal tenant (for example a tenant management tool), it needs to open separate database connections to each tenant. Table 2 summarizes how meta data and content is stored and accessed for the three types of tables.

As a further measure to guarantee tenant isolation, each tenant is assigned to its own transaction domain. This ensures that a transaction is restricted to one tenant and that that a single transaction cannot span multiple tenants. Therefore, we only allow distributed transactions within one tenant but not across tenants.

Having different disk volumes for different tenants makes it easier to support tenant life-cycle management, such as tenant copy, tenant relocation, or tenant deletion. When relocating a tenant to a different IMCE database server in the same IMCE system, the tenant-specific volume can be detached from the original server and attached to the new server.

For moving or copying tenants between different IMCE systems, the problem arises how shared meta data should be handled. For such a cross-system tenant move, it is ensured that the tenant data to be moved is consistent with central meta data in the new system. Tenant creation is implemented by cloning a template tenant.

Database clients need multiple database connections if they need to access more than one tenant. However, sometimes applications need to combine tenant-dependent tables with tenant-independent tables (stored in the system tenant) in one query – for example in a join operation, a sub-query, or a union. To process this type of queries, the tenant servers for normal tenants have indirect read access to tenant independent tables. A database client that is connected to the database server of a normal tenant may combine tenant-independent tables and tenant tables in the same query. If a tenant server receives such a query, it delegates the corresponding operations to the database server of the system tenant, combines the results with local results, and returns them to the database client. Therefore, it provides federation capabilities between the local and the system tenant. This way, the database clients need not be aware of the fact that the system tenant is involved.

Because the system tenant belongs to a different transaction domain, a query that involves access to tenant-independent tables is executed by running two different transactions. Therefore, transparent access to tenant-independent tables by a normal tenant is limited to read-only operations. If a database server assigned to a normal tenant receives a request to modify the content of tenant independent-tables, it reports an error. A database client that needs to write tenant-independent tables needs to open a connection to the system tenant server.

As introduced above, if meta data is created in a normal tenant, it is stored in the tenant as private meta data which is not accessible by other tenants. When reading meta data in the context of one tenant, the result is calculated as the union of central meta data and tenant-private meta data. This is done by the meta-data managers and is hidden from all other IMCE components. It is possible to use private meta data for tenant-specific extensions of centrally defined meta data. Tenant-specific column extensions are implemented by creating new columns (using ALTER TABLE) in the tenant-private meta data. These additional fields only exist in this specific tenant.

6 Conclusion

In this paper, we highlighted the multitenancy features of the in-memory computing engine. The in-memory computing engine is SAP's main-memory database management system. Its target hardware is a cluster of blades, where each blade can hold up to 2 Tb of main memory and up to 64 physical cores. The IMCE has a row store and a compressed column store, supports insert-only data management, data partitioning and distribution across blades, as well as parallel query processing.

Regarding multitenancy, the IMCE provides support at two conceptual levels: at the database level, by tenant separation and life-cycle management, and at the application level by providing capabilities for meta-data and data sharing. In IMCE, distribution and

multitenancy are closely related. The multitenancy implementation builds on meta-data replication features of distributed IMCE instances.

Acknowledgements

We would like to thank Meinolf Block for his valuable input to this document. Furthermore, we would like to thank the experts of TREX, PTime, LiveCache, and MaxDB, who built the SAP in-memory computing engine.

References

- [AJPK09] Stefan Aubach, Dean Jacobs, Jürgen Primsch, and Alfons Kemper. Anforderungen an Datenbanksysteme für Multi-Tenancy- und Software-as-a-Service-Applikationen. In *BTW*, pages 544–555, 2009.
- [BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores. In *Proc. SIGMOD*, pages 283–296, 2009.
- [ByD] SAP Business ByDesign. <https://www.sme.sap.com>.
- [CDG⁺08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions Computer Systems*, 26(2), 2008.
- [CRS⁺08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!’s Hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, 2008.
- [DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jambani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s Highly Available Key-Value Store. In *Proc. SOSP*, pages 205–220, 2007.
- [FK09] Daniela Florescu and Donald Kossmann. Rethinking Cost and Performance of Database Systems. *SIGMOD Record*, 38(1):43–48, 2009.
- [GMS92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.
- [GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [HPP10] HP ProLiant DL980 G7 Server, 2010. http://h18004.www1.hp.com/products/quickspecs/DS_00190/DS_00190.pdf.
- [HR01] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, 2001.

- [JA07] Dean Jacobs and Stefan Aulbach. Ruminations on Multi-Tenant Databases. In *Proc. BTW*, pages 514–521, 2007.
- [KGT⁺10] Jens Krüger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Franz Faerber. Optimizing Write Performance for Read Optimized Databases. In *Proc. DASFAA*, pages 291–305, 2010.
- [LKC01] Juchang Lee, Kihong Kim, and Sang Kyun Cha. Differential Logging: A Commutative and Associative Logging Scheme for Highly Parallel Main Memory Databases. In *Proc. ICDE*, pages 173–182, 2001.
- [Lor77] Raymond A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1):91–104, 1977.
- [LSF09] Christian Lemke, Kai-Uwe Sattler, and Franz Färber. Kompressionstechniken für spaltenorientierte BI-Accelerator-Lösungen. In *Proc. BTW*, pages 486–497, 2009.
- [MSLR09] Olga Mordvinova, Oleksandr Shepil, Thomas Ludwig, and Andrew Ross. A Strategy for Cost efficient Distributed Data Storage for In-Memory OLAP. In *Proc IADIS AC*, pages 109–117, 2009.
- [OAE⁺09] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMclouds: Scalable High-Performance Storage Entirely in DRAM. *Operating Systems Review*, 43(4):92–105, 2009.
- [PI] SAP Process Integration. <http://www.sap.com/platform/netweaver/components/pi/index.epx>.
- [Pla09] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proc. SIGMOD*, pages 1–2, 2009.
- [Rah94] Erhard Rahm. *Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison-Wesley, 1994.
- [RR00] Jun Rao and Kenneth A. Ross. Making B⁺-Trees Cache Conscious in Main Memory. In *Proc. SIGMOD*, pages 475–486, 2000.
- [Sto86] Michael Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.
- [WPB⁺09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using On-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.

Available-To-Promise on an In-Memory Column Store

Christian Tinnefeld¹, Stephan Müller¹, Helen Kaltefleiter¹, Sebastian Hillig¹,

Lars Butzmann¹, David Eickhoff¹, Stefan Klauck¹, Daniel Taschik¹,

Björn Wagner¹, Oliver Xyländer¹, Cafer Tosun², Alexander Zeier¹, and Hasso Plattner¹

¹ Hasso Plattner Institute, University of Potsdam, August-Bebel-Str. 88, 14482 Potsdam, Germany, Email: (firstname.lastname)@hpi.uni-potsdam.de

² SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany, Email: (firstname.lastname)@sap.com

Abstract: Available-To-Promise (ATP) is an application in the context of Supply Chain Management (SCM) systems and provides a checking mechanism that calculates if the desired products of a customer order can be delivered on the requested date. Modern SCM systems store relevant data records as aggregated numbers which implies the disadvantages of maintaining redundant data as well as inflexibility in querying the data. Our approach omits aggregates by storing all individual data records in an in-memory, column-store and scans through all relevant records on-the-fly for each check. We contribute by describing the novel data organization and a locking-free, highly-concurrent ATP checking algorithm. Additionally, we explain how new business functionality such as instant rescheduling of orders can be realized with our approach. All concepts are implemented within a prototype and benchmarked by using an anonymized SCM dataset of a Fortune 500 consumer products company. The paper closes with a discussion of the results and gives an outlook how this approach can help companies to find the right balance between low inventory costs and high order fulfillment rates.

1 Introduction

There is an ongoing discussion in the database community to what extent applications can benefit from a database management system (DBMS) that exactly suits their needs. One central paper in this discussion is written by Stonebraker and Cetintemel [Sc05] who argue that applications such as text search, scientific applications, data warehousing, and stream processing can benefit from a performance, maintenance, and functionality perspective by using application specific database engines. As stated by Krueger et al. [KTG⁺10], we think that this statement is also true for the domain of traditional enterprise applications which we want to exemplify in this paper with the ATP application.

ATP provides a checking mechanism to obtain feasible due dates for a customer order. This is done by comparing the quantities of products which are in stock or scheduled for production against the quantities of products which are assigned to already promised orders [Dic05, SZ03]. A common technique in current SCM systems is using aggregated values for keeping track of the different quantities, which results in having a separate aggregate for each different product. This means that e.g. a new product in stock would

increase the value of such an aggregate while the assignment of products to a confirmed customer order would decrease it. Although the use of aggregates reduces the necessary amounts of I/O operations and CPU cycles for the single ATP check itself, it introduces the following disadvantages:

Redundant Data. One problem that arises in association with materialized aggregates is the need for data replication and therefore for complex synchronization strategies [Pla09]. In order to preserve a consistent view on the data across the whole system, every write operation has to be propagated to all replications. Even if the updates are triggered immediately, they still imply delays causing temporary inconsistencies. Additionally, even if the amount of I/O operations and CPU cycles is reduced to a minimum for the check itself by using aggregates, the overall sum of needed operations might be higher due to synchronization as well as maintenance and costly back calculation of the aggregates.

Exclusive Locking. A related issue consists of locking for update operations. All modifications to an aggregate require exclusive access to the respective database entity and block concurrent read and write processes. The downside of locking is obvious, as it queues the incoming requests and affects the performance significantly in case of a highly parallel workload.

Inflexible Data Querying. Evidently, the gain in performance concerning isolated queries comes at the cost of less flexibility. For ATP systems in particular, this fixedness poses major restrictions. The rolled up data structures are tailored for a predefined set of queries. Unforeseeable operations referring to attributes that were not considered at design time cannot be answered with these pre-aggregated quantities. Those attributes include for instance shelf life, product quality, customer performance and other random characteristics of products, orders, or customers. Additionally, due to the use of aggregates the temporal granularity of the check is fixed. Once the aggregates are defined and created based on e.g. the available quantities per day, it is not possible to perform ATP checks on an hourly granularity.

Inflexible Data Schema Extensions. The previously mentioned inflexibility of not being able to change the temporal granularity of a single check indicates another related disadvantage: the inability to change the data schema once an initial definition has been done. The change of the temporal check granularity or the inclusion of a previously unconsidered attribute is only possible with a cumbersome reorganization of the existing data.

No Data History. Maintaining aggregates instead of recording all transactions enriched with information of interest means to lose track of how the aggregates have been modified. In other words, no history information is available for analytics or for rescheduling processes.

As stated above, the evolution of business needs indicates an increasing relevance of sophisticated analytical applications. In order to realize immediate answer to arbitrary analytical queries without long lasting ETL processes, raw data in a format that enables

on-the-fly processing is needed. Facing these demands, recent trends are heading towards column-oriented in-memory databases. In-memory databases keep all data in main memory and therefore facilitate fast and random access. To be able to hold billions of data records in memory, high compression rates are mandatory. By storing the data column-wise, generally suitable compression techniques can be applied. The main reason for good compression results achievable in column-stores is the similarity of the entries in a column, since the compression ratio is often dominated by the number of distinct values [KGT⁺09].

Another advantage coming along with column-stores is that many operations, notably aggregations, equi-joins, and selections, can be performed directly on compressed data [AMH08, GS91]. Consequently, fewer entries have to be decompressed for reconstruction of tuples. This strategy called lazy decompression [CGK01] helps to save CPU cycles for decompression. Reducing computing time is particularly in in-memory databases highly relevant, because I/O costs are extremely low, so that CPU time influences the overall execution time significantly [HLAM06]. This technique is especially beneficial in the context of insert-only as updates do not directly require a decompression of already stored values, but result in appending new values for already existing tuples. Furthermore, the read-optimized columns can go along with a smaller, write-optimized so called delta store which is used for updates.

Consequently, in column-oriented in-memory databases aggregates can be calculated by processing the appropriate column without the need to read the entire table from disk or decompress it in advance. Thus, column-oriented database systems can operate on huge datasets efficiently and thereby comply with the requirements of an OLAP system very well [BMK99]. They also bring along a new level of flexibility, since they are not confined to predetermined materialized aggregates. Write operations are not limited by the read-optimized data structure as they are performed in the delta store.

The remainder of this paper is organized in the following way: Section 2 presents the involved concepts which are needed for executing an ATP check on a columnar database. This includes aspects such as data organization, the check algorithm itself, and how to deal with concurrency. Section 3 includes the description of a prototypical implementation of the concepts and corresponding benchmarks which were done on an anonymized SCM dataset of a Fortune 500 consumer products company. The prototypical implementation has been done in the context of a joint research project between the Hasso Plattner Institute and SAP. Section 4 discusses the possible business implications of this approach by listing new or improved functionalities in the context of ATP. Section 5 concludes with a discussion of the results and an outlook how this approach could be used for taking ATP to the next level by providing a profit-maximizing ATP checking mechanism.

2 Involved Concepts

With a general overview of the ATP check and its limitations with the use of aggregates, this section provides the underlying concepts of the prototypical ATP application based

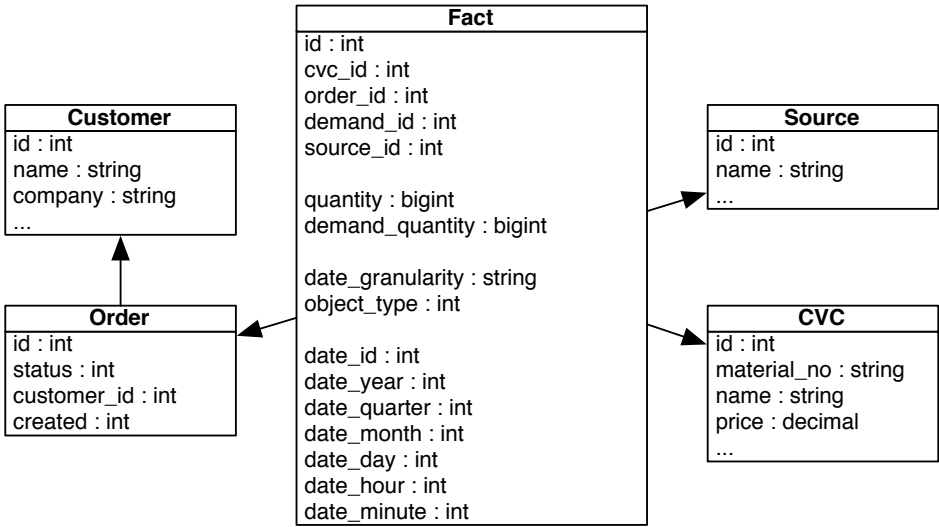


Figure 1: Simplified physical schema as used in prototype

on a columnar, in-memory database. Subsection 2.1 introduces the main classes in the ATP data model and their physical representation in the database. The subject of Subsection 2.2 are two algorithms to calculate the due dates focusing on their applicability on a column-oriented database system. In Subsection 2.3 finally the strengths and weaknesses of different strategies to handle concurrent ATP requests are discussed.

2.1 Data Organization

In this subsection, a simplified data model sufficient for a basic understanding of the prototype is presented. The information relevant to an ATP system are primarily line items of sales orders also referred to as customer demands, delivery promises or conducted outputs, the stock level, and planned production inputs. These types of transaction data are consolidated in one table with an object_type column for identification, forming the fact table in the star schema [Mar04].

Essentially, an entry in the fact table indicates how many items of a product leave or enter the stock on a specified date. The products are listed by their characteristic value combinations (CVC), to be uniquely identified. In the fact table, there are two quantity columns. This is due to the fact that the customer demands indicate what was ordered and do not represent planned stock movements, which are stored in the promises. To be able to provide information about the inventory, only planned and conducted stock movements but not the customer demands have to be added up. For this reason, the customer demands have a separated quantity column, the demand_quantity. This way, they do not have to be filtered when aggregating the stock movements in the quantity column. Recurring data,

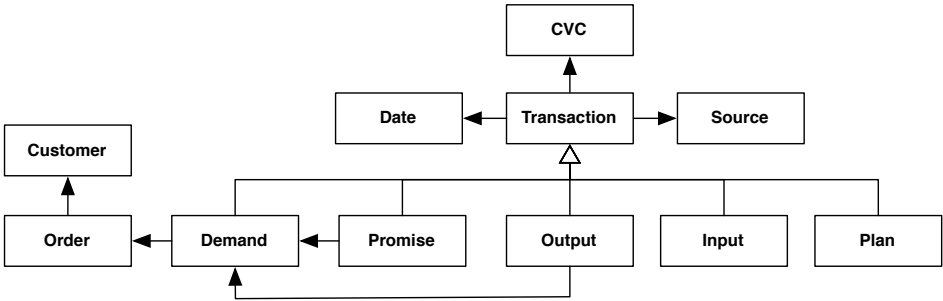


Figure 2: Simplified ERM of the ATP process

id	date_id	cvc_id	demand_id	demand_quantity	quantity	object_type
5	1286668800	1	5	-45	0	3
6	1286668800	1	5	0	-45	1
7	1286409600	1	0	500	500	2
8	1286323200	2	3	-10	0	4

Table 1: Fact table extract

such as the customer who ordered the product or the source of an input, be it a production plant or supplier is stored in dimension tables.

To avoid expensive joins with a potentially large Date table, this dimension is de-normalized, accepting a certain degree of redundancy. The physical data model as implemented in the prototype is shown in Figure 1. For the sake of clarity, the following examinations abstract from this and other optimizations. The business objects used in the application are illustrated in Figure 2.

As a typical feature of a star schema, most columns in the fact table are foreign keys to dimension tables. It can be expected that the content of the majority of dimension tables is static. The entries in the Date table will only be updated, when the fiscal year comes to an end and the planning horizon changes. Just as reasonable is the assumption that the content of the CVC and Source tables is constant. Changes to these tables are only necessary, if the company introduces a new product, puts a production plant into operation, or goods are purchased from a new supplier. As a consequence, the total volume of data can be reduced by storing recurring information in dimension tables.

In Table 1 an extract from the fact table, with the redundant date specifications date_year to date_minute as well the columns order_id, source_id, and date_granularity left out, is provided. The first row with id 5 represents a customer demand with the computed delivery promise in the second row. This relation can be seen by the foreign key demand_id, which is set to 5 in the promise entry and hence points to the demand row. Besides, the object types 3 and 1 identify these rows as demand and promise. Since they correspond in quantity and date, the request can be fulfilled in time.

The third column is an input, characterized by the object_type 2, of 500 items to the same product that was requested by the afore mentioned demand. Inputs do not refer directly to a customer demand in order to stay as flexible as possible when it comes to unexpected loss of stock and a redistribution of resources is required. So, the foreign key demand_id is a null pointer. One might wonder, why the quantity is replicated in the column demand_quantity. This way, we retain the option to run the ATP check against the requested instead of the promised quantities and thereby favor already booked orders over new ones.

Object_type 4 identifies withdrawals from stock made to accomplish a customer order. Basically, a promise is turned into such an output, as soon as the items are taken from the warehouse and on their way to the customer. To save the connection between demand and outputs, the outputs also store the id of the demand in the dedicated foreign key column. The last row in Table 1 is an example for an output. The demand it refers to, the fact with id 4, is not listed in the extract.

2.2 ATP check Algorithms

The core functionality of an ATP application is the determination of earliest possible delivery dates based on the inventory. The available capacities accrue directly from stock, inputs, and previously confirmed orders. This subsection suggests and evaluates two solutions to this problem, addressing the decisive ATP requirements and the characteristics of our prototype. The two algorithms are equal in terms of correctness and complexity but differ in performance depending on the underlying architecture and dataset.

2.2.1 Candidate Check

The *candidate check* constitutes the first alternative to compute due dates. The name candidate check is derived from the supposed promises, the so called candidates, which are created in course of the check. Basically, the algorithm operates on a chronological list of dates associated with the aggregated inputs and outputs for the particular dates. Thereby, dates with a total quantity of zero will be omitted to reduce the runtime. Such (date, quantity) - tuples will be referred to as *buckets* and chronological lists of buckets as *time series*. Apart from the time series, the algorithm maintains a list of candidate buckets as temporary promises, also sorted by date in ascending order. As an interim promise, a candidate indicates the quantity that is allocated on the specified date for the current demand. The word temporary is meant to point out the potential deletion or reduction of the candidate while iterating over the remainder of the time series. Those candidates that can be retained until the algorithm terminates, will be written as promises to the database. If there is only one candidate with the desired quantity on the desired date, the order will be fulfilled in time.

In the following, details of the algorithm, particularly the determination of candidates, will be explained. To identify candidates, the total stock level for each date is required. Therefore, a new data structure is deduced from the original time series, henceforth termed

date	aggregated quantity	accumulated time series
MO	2	2
TU	1	3
WE	-2	1
TH	1	2
FR	2	4

Table 2: Accumulated time series

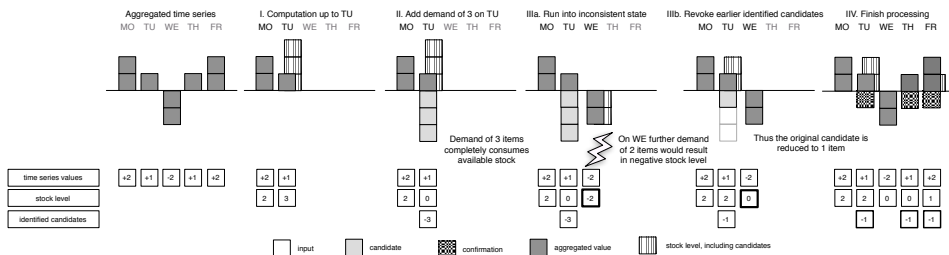


Figure 3: Candidate check example

accumulated time series as can be seen in Table 2.

From the very beginning of the planning horizon, all quantities are added up to the desired date. Reaching that point in the time series, the creation of candidates starts. If the accumulated quantity is positive, the first candidate will be initialized. Generally, the quantity of a candidate is limited to the stock level of the respective date, in this case the desired date. The maximum quantity of a candidate is logically the desired quantity. For further processing, the accumulated quantity is reduced by the quantity of the new candidate. As long as the desired quantity has not been completely allocated, candidates will be created while processing the time series.

When the stock level drops below zero due to other promises or a decline in production, the list of candidates has to be corrected. Thereby, the candidate quantities are deallocated until the stock level returns to zero or the list is exhausted. To ensure best possible delivery dates, the list is updated descendingly, removing the latest buckets first. The entire candidate algorithm is formally described in Listing 1. To highlight the essential control flow, the readjustment of the candidate list in case of a negative accumulated quantity is not listed on instruction level but hidden in the method *truncate_qty*, which is invoked on the candidate list, cf. Listing 1 Line 26.

To improve the understanding of the algorithm, a walk-through with a concrete example is undertaken. The available stock including one output, represented in the first diagram in Figure 3, and a new order of three items for Tuesday are the starting point for this excursion. The first diagram shows the time series with a negative bucket on Wednesday. This might be confusing in the first place, as it appears to be an overbooking. In fact, it is not an overbooking, which becomes obvious when calculating the accumulated time series

as seen in Table 2. The three inputs on Monday and Tuesday compensate the outputs on Wednesday.

```
1 def candidates(time_series, desired_date, desired_qty):
2     candidates = []
3     acquired = 0
4     accumulated = 0
5     for date, qty in time_series:
6         accumulated += qty
7         # do not start before desired_date
8         if date < desired_date:
9             continue
10        if accumulated > 0:
11            wanted = desired - acquired # pending quantity
12            if wanted > 0:
13                if accumulated >= wanted:
14                    # total covering of wanted quantity
15                    candidates.append((date, wanted))
16                    acquired = desired
17                    accumulated -= wanted
18                else:
19                    # partial covering
20                    candidates.append((date, accumulated))
21                    acquired += accumulated
22                    accumulated = 0
23            elif (accumulated < 0) and (len(candidates) > 0):
24                # acquired too much, give back until
25                # accumulated is not negative
26                truncate_qty(candidates, accumulated, acquired)
27    return candidates
```

Listing 1: Candidate Check Algorithm

Going one step back, the accumulated time series is built up to Tuesday. As already mentioned and reflected in Line 8 in Listing 1, the creation of candidates starts at the desired date, which is Tuesday in this example. The accumulated quantity, which results from the Monday and the Tuesday bucket, is three corresponding to the desired quantity. In compliance with Lines 15 to 17 in Listing 1, a candidate for Tuesday with three items is appended to the still empty candidate list, the already acquired quantity is set to three, and the accumulated quantity is cut to zero. This new candidate can be seen in diagram II in Figure 3. On the next day, two items leave the stock due to the output. Consequently, the accumulated quantity drops below zero as can be seen in IIIa. To compensate the overbooking, two items have to be given back. Therefore, the candidate list is processed backwards. Since there is only one entry holding three items, this candidate will be truncated to one, cf. diagram IIIb in Figure 3, and the acquired quantity will be reduced accordingly. Proceeding the same way, a candidate for Thursday with one item and one for Friday covering the last pending item will be appended to the candidate list. At the end of the planning horizon, on Friday, the accumulated quantity representing the stock level is still positive so that the three candidates will be returned as promises.

date	input	promises	net
MO	2	-1	1
TU	2	0	0
WE	1	-3	0
TH	3	-2	1
FR	1	0	1

Table 3: Net time series aggregates

With regard to the fact that per date either a candidate is created or existing candidates are removed, the candidate check features a linear complexity. Besides, one can easily see that this algorithm performs best in case of sufficient stock, because only one candidate is created and no updates are required.

2.2.2 Net Time Series Check

As stated above, there is a second approach, the *net time series check*, leading to the same promises and showing the same complexity. Since this algorithm was not integrated into the prototype for technological reasons, only the main idea will be outlined. The starting point for the net time series check are two time series, aggregating disjoint sets of fact table entries. One time series adds up the inputs and the other one all confirmed and conducted promises.

In a next step, bucket by bucket the aggregated promises are matched to the quantities of the input time series. A bucket from the promise time series consumes primarily items from the input bucket of the same date. If this corresponding input bucket does not satisfy the promise bucket, first earlier input buckets and after that later ones will be emptied. The resulting net time series represents the resources that are available at the time of the ATP request, cf. Table 3, and gives this algorithm its name. When the net time series is set up, the new demand acts just like the promise buckets and takes out the requested quantity from the net time series. Promises are created based on the obtained input buckets, .

Whereas the candidate check has to consider the whole planning horizon independent on the stock level, the net time series check can under certain conditions reach minimal computing times. If there are no promises or if they all fall into a small number of buckets and sufficient inventory is available, the algorithm only has to match a few buckets. Under the premise that they can even be fulfilled out of the preferred input buckets, only a fixed number of operations, which is determined by the number of promise buckets, is required and a constant complexity is reached. In such scenarios, the net time series check outperforms the candidate algorithm. This scenario is unlikely in production though, as it requires all confirmed promises to fall onto a few condensed dates.

2.2.3 Comparison

To sum up, the two presented algorithms deliver optimal promises to the customer and both vary in performance depending on the characteristics of the dataset. So, from a merely logical point of view both alternatives are equal. However, taking technological aspects into account, major differences can be identified.

```
SELECT SUM(Fact.quantity), MAX(Fact.date_id)
FROM Fact
WHERE Fact.cvc_id = 1
GROUP BY Fact.date_year, Fact.date_month, Fact.date_day
ORDER BY MAX(Fact.date_id)
```

Listing 2: Candidate check aggregation

The descriptions above start with the initial time series already available. The creation of these data structures has not been treated so far. In fact, it is the most time consuming part in the overall ATP check, because the raw data has to be aggregated on-the-fly. To set up the time series for the candidate check, all inputs and promises of the specific product are added up grouped by the selected time granularity. The resulting database query for the granularity level day is shown in Listing 2. The maximum of all dates per bucket is selected, because depending on the granularity several timestamps belong into one aggregation class. It is necessary to find the latest timestamp of all aggregated records, to make sure that at this point in time all movements have already been issued.

For the net time series check, the situation is more complicated. Two separated time series are necessary, which can basically be achieved in two different ways. The first option would be to add an additional group-by attribute, the object-type respectively, to the query for the candidate check. The downside of this method lies in the structure of the result set, which includes the buckets for both time series. Thus, new empty time series are created and while iterating over the result set the buckets are inserted either into the input or the promise time series. Furthermore, with an increasing number of group-by attributes the query execution time increases substantially. Alternatively, the two time series can be extracted from the database in separate queries by adding another predicate for the object type. This approach obviously requires two full table scans and thus does not present a feasible solution, especially when the application has to deal with several millions of records.

Another disadvantage, which applies to both query variants equally, is the quantity of data to be transferred from the database to the application layer, as inputs and promises are not consolidated into one time series. Being aware of the disadvantages of the net time series check with respect to the database queries, the decision in favor of the candidate check becomes more transparent.

2.3 Concurrency Control

The algorithms listed in Subsection 2.2 focus on the execution of one ATP check in isolation. They do not factor in the difficulties caused by multiple parallel checks referring to the same product and consequently accessing the same data. The management of concurrent processes indeed belongs to the main challenges in parallel environments. Particularly in an ATP system, data correctness and consistency at any time is an essential requirement to avoid wrong promises.

To be precise, there is a temporal gap in between reading the current stock levels from the database and writing a promise in the end based on those results. In the meantime another process, proceeding from the same inventory, might have calculated a delivery date and booked resources that are necessary to fulfill the first request. As a consequence, the same products are promised twice causing an inconsistent state in the database. Such anomalies are a serious problem in an ATP system. A company might take severe damage from dealing with the resulting effects including angry customers, contractual penalties, costs for acquiring substitute products, and so on. In current ATP systems, the common practice is to serialize the execution of concurrent requests by locks. Our prototype allows for choosing out of three strategies suitable for different situation. These three approaches will be elaborated in this section, benchmarking results will be presented in Section 3.

2.3.1 Exclusive Lock

A naive but secure way to preserve consistency constraints is as mentioned above to lock critical data. In this context, it means to grant to one process exclusive access rights to the entire set of fact table entries for the desired product. Since always the whole planning horizon has to be checked, simultaneous checks on different dates are unrealizable. The first incoming process acquires the lock, queries the database, calculates the delivery dates, writes the promises back to the database, and finally releases the lock for the next process to start. Apparently, this locking policy, termed exclusive lock, involves superfluous latencies in case of sufficient stock. If there were several hundreds of incoming requests for one product per minute, a sequential schedule would lead to response times that exceed the limit of tolerance.

2.3.2 Optimistic

Whereas the exclusive lock queues incoming requests for the same product, the second solution, an optimistic strategy, enables parallel execution without blocking. Theoretically, the optimistic mechanism allows for as many parallel requests as cores available and therefore scales linearly with hardware resources. For now, it seems as if the gain in scalability implies a certain staleness of the data. Indeed, without modification of the ATP check, violations to the consistency may occur.

This modification consists of a consistency check after the original candidate check. A process has to verify the correctness of its result concerning the new stock level. The term

optimistic expresses the nature of this strategy presuming a balanced stock situation. To avoid a second full table scan, the maximum row id of the fact table identifying the most recently written record is retrieved from the database in advance. Afterwards, the check is performed based on the stock level up to this id. It must be mentioned that the row id is a continuously incrementing counter, which facilitates absolute ordering of fact table entries by insertion time.

If the candidate check results in a complete or at least in a partial delivery, the promises are written to the database and a consistency check will be performed. For this purpose, all fact table entries with an id higher than the initially determined maximum id are retrieved. The result set contains exactly those inputs and promises that were recorded during the check. The records are successively included into the time series that comprises the entries up to the maximum id. In cases of outputs, for instance corrections to planned inputs or promises, the stock will be checked for overbooking and the invalid quantity will be saved. If the record is a promise related to the current demand, the system will use its quantity to compensate the overbooked quantity. So the promise will be either deleted or reduced. When all records are processed, a new ATP request will be triggered with the total rebooked quantity. Evidently, this conflict resolution procedure can end up in an infinite loop. In order to enforce a termination, one could define a maximum recursion depth and switch to the exclusive lock, once this depth is reached.

To sum up, the optimistic approach dispenses with locks, unless the inventory is close to exhausting and many parallel requests are competing for the last items. Though, as long as sufficient capacities are available, the avoidance of locks can be fully leveraged by providing appropriate hardware resources.

2.3.3 Instant Reservation

The third idea arises from the drawbacks of the two afore mentioned ones. In general terms, it works similarly to the optimistic strategy without the need for conflict resolution. The key to this enhancement lies in blocking the desired quantity before the check is started. So, this approach is called instant reservation. To reserve the requested quantity, a promise complying with the customer demand is written directly to the database. For the candidate check, only the fact entries up to this promise are aggregated so that the check will not be manipulated by its own reservation.

Once the result is computed, it is compared to the initially recorded promise. If they do not correspond to each other, an adjustment will follow. At this point, it must be mentioned that this process is totally transparent to the user of the system. The reservation promise will not be passed on to the user, unless it corresponds to the calculated one. Otherwise, the updated promises will be delivered.

Reviewing the sequence of steps, one might have noticed that the reservation can cause an overbooking. But through the comparison in the end, each process clears up self-inflicted inconsistencies. Concurrent processes include earlier reservations in their calculations. Accordingly, it is guaranteed that they do not allocate resources needed to fulfill other earlier processes. Even if the earlier processes have not finished their check yet, the desired

quantities are blocked by the reservation.

However, in one specific case, this approach does not produce an optimal utilization of available resources. Given an empty stock and two incoming requests with the same quantity, the first one writes its reservation and initiates the aggregation excluding the reservation. Before the second request writes the reservation promise, an input is recorded with exactly the same quantity like the demands. This input is not seen by the first process so that the request will be rejected.

Therefore, it would be desirable that the input will be used to satisfy the second process. This process, however, includes both, the input and the reservation of the first process. Thus, it also faces an empty stock and refuses the customer request. Being aware of this problem, one has to decide, whether he will accept it and receive the benefits arising from the renouncement of locks as well as conflict resolution. More detailed information about the performance of all three mechanisms will be provided in Subsection 3.1.

3 Prototypical Implementation and Benchmarking

As mentioned in Section 1, the presented prototypical implementation has been done in the context of a joint research project between the Hasso Plattner Institute and SAP. The used database system is based on proprietary SAP DBMS technology and is referred to as relational, column-oriented, in-memory DBMS in the remainder of this paper.

Our prototype ATP had to tackle the following architectural requirements: Multiple concurrent orders have to be possible and are to be handled without risking inconsistency. The prototype renounces the limitation of static buckets in current ATP systems, enabling distinctions in delivery date granularity on a per-demand basis.

The implementation we set out to build was to be optimized for columnar, in-memory storage [Hen10]. In order to provide fast order confirmation, our implementation also exploits parallel execution potential within the concurrency control strategies. It provides special views for analytical queries which allowed us to specify key figures upfront, yielding improved performance in analytical queries, particularly those important to create the time series for a candidate check. The application logic is implemented in Python, accessing the database with via its SQL interface. Computationally intensive parts have been ported to C++ and are imported as compiled modules.

3.1 Benchmarks

The benchmarks were conducted on a 24 core Xeon (four physical cores each comprising six logical cores) with 256 GB of main memory. This server was used for both, application and database, eliminating network performance as a factor in accessing the database. Furthermore, in our benchmarks we disabled logging of transactions to prevent hard disk transfer speed from posing as a bottleneck in our scenarios.

Our benchmarks were conducted on a dataset that was derived from a live ATP system of a Fortune 500 company. Today’s ATP systems do not support analytics on top of ATP checks and only keep a short horizon of data as an active dataset and discard data after a few days. Thus, we generated a dataset spanning 3 years from a set of records covering 3 months of planning horizon. We ran our benchmarks on datasets ranging from 1 to 64 million active transaction items, where every order has 5 to 15 line items. Other dimensions were not scaled with the number of transactions, simulating different company sizes.

The immediate goal of our benchmarks is to show the feasibility of our prototype and proposed data structures. The exclusive strategy provides a comparison with existing systems from an algorithmic standpoint - it reflects how checks in modern ATP systems are conducted with the only difference being the data storage.

3.1.1 Dataset Size

The first benchmark serves as a proof of concept of the database architecture applied in our system. On a varying dataset size, the same operations are performed. In each run, 10 concurrent processes execute 400 checks. The critical part concerning the overall runtime of an ATP check is the aggregation query. The time spent in the application or to be exact the candidate check is supposed not to be affected by the dataset presuming a constant distribution of fact entries over the time. In this case, the size of the time series is determined only by the chosen granularity, which will not be changed so that the application part is constant.

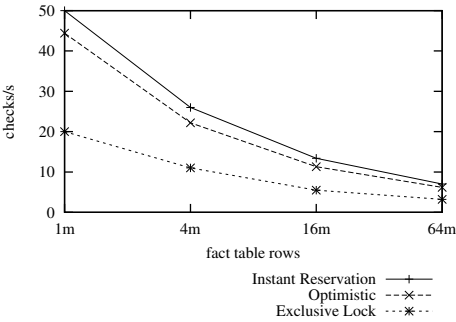


Figure 4: Varying dataset size

To conclude, the expected outcome of this benchmark is a linear relation between the dataset size and the query execution performance. Figure 4 displaying the throughput of checks explicitly reflects this trend. For the remaining experiments in this paper we work on 64 million records item which represents three years of operations in a large company.

3.1.2 Concurrency Control - Single Check

The following experiments directly compare the three concurrency control mechanisms introduced in Subsection 2.3. For this purpose, at first a single check is executed in isolation to evaluate the overhead added by the concurrency control. To recap, the exclusive lock only wraps the candidate check with the acquiring and releasing of a lock on product level. Both are atomic operations and do not involve database access. So, the overhead is supposed to be negligible.

In contrast, the optimistic approach requires a consistency check, which might induce a loop of conflict resolutions, with a maximum recursion depth though. The instant reservation mechanism neither uses locking nor consistency checks. Instead, it writes the promise first and has to validate it in the end. In case of a mismatch with the calculated delivery confirmation, an additional write operation to correct the promise is triggered. In Figure 5, the elapsed time split into read, write, and computation is illustrated. As expected, the optimistic check performs worst in this experiment, because it has to retrieve the recent fact entries to control the stock level after booking.

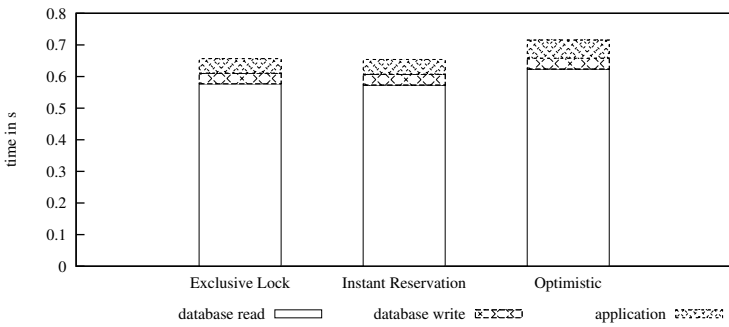


Figure 5: Single check

Since the inventory has not changed during the check and enough items were available, a conflict resolution has not happened. Neither a promise adjustment in the instant reservation run was necessary. Hence, the overhead is minimal for the two strategies that dispense with locking. The process of the exclusive lock is independent on the scenario and does not prolong the check anyway. Nevertheless, the scenario was not constructed to show the best case performance of the any strategy but rather to emulate the most common conditions with sufficient stock.

3.1.3 Concurrency Control - Throughput

After comparing single check times, the effectiveness of the presented techniques when facing parallel requests is measured, as it is the primary reason for putting so much emphasize on this topic. The prevailing KPI to assess concurrency control strategies is the throughput, in our case the number of accomplished checks per second. The setup for

this experiment consists of 1000 checks to be executed on the machine specified above by a varying number of processes representing the degree of parallelization. In the first experiment, the checks refer to different products.

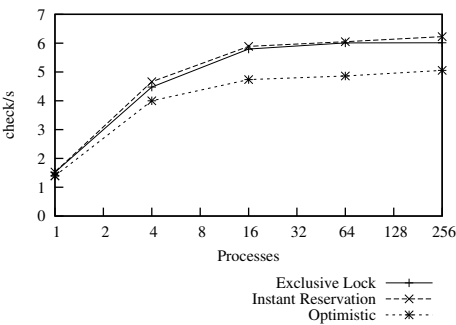


Figure 6: Disjoint CVC access

On the basis of the single check times and the hardware capacities, conclusions about the throughput can easily be drawn. The exclusive lock allows parallel requests on different products, so do the optimistic and the instant reservation approaches. Logically, the throughput scales linearly with an increasing number of processes. Since there is only a limited number of physical and logical cores, the increase of the curves flattens when the hardware is fully exploited, as can be seen in Figure 6.

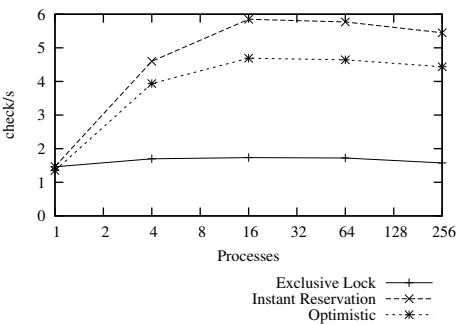


Figure 7: Single CVC access

For concurrent checks on one single product, another behavior is to assume. No matter how many processes are used to carry out the 1000 checks, only one process at a time can operate on the requested product in case of the exclusive lock. The logical consequence would be a constant throughput, which could be verified experimentally in our measurements, cf. Figure 7. This benchmark further gives evidence for the scalability of the two other alternatives in case of concurrent checks on one product. Those scenarios heavily benefit from locking-free concurrency control, whereas the exclusive lock enforces a sequential execution.

3.1.4 Write-Intensive Operations

Column-stores are particularly tailored for analytical queries. The workload of an ATP application is not limited to availability checks only but includes regular write operations as well. New orders and delivery promises have to be saved with every order. All new inputs and changes in inventory and production result in update and insert operations. The prevailing solution to handle a mixed workload consists of the division of the database into two parts, a read-optimized and a write-optimized store, the delta store, as briefly touched on in Section 1. The read-optimized store contains a snapshot of the database at a pre-established point in time. All incoming modifications are written to the delta store [KHK80]. The idea of organizing the data in two isolated stores dates back to the sixties. Early studies propose to regularly consolidate all modifications in the write-optimized store into the read-optimized store [SL76]. This so called merge process is the only procedure that modifies the read-optimized store. Since there is no need for maintainability or updatability, it can store the data most suitable for fast read access.

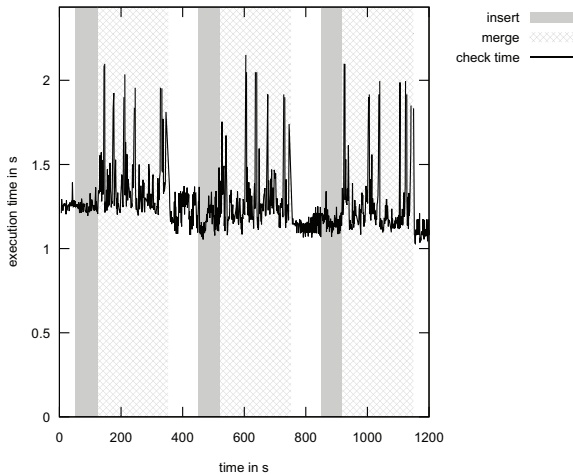


Figure 8: Performance during merge

In the delta-store, fast inserts can be enabled by not using the compression and keeping the data unsorted. The obvious effect is that ATP queries slow down, when the delta store reaches a certain size, since both stores need to be considered. In the light of this performance loss, a naive suggestion would be an eager merge policy minimizing the response times of analytical queries. However, the merge process itself is a complex procedure consuming a considerable amount of system resources. So, merge scheduling strategies have to deal with the tradeoff between merging often to keep the write-optimized store small and merging rarely to reduce the influence of the merge on the regular operation. To get an impression on how the merge process affects the ATP query execution, a long-term benchmark has been run on a dataset of 64M records. One process continuously executes ATP checks and another one inserts in cycles 1000 fact entries that are merged instantly to the read-optimized store. The curve in Figure 8 shows the total time spent on one check

while the second process is in insert, merge, and idle phases and highlights the importance of a smart merge policy.

4 Business Implications

After describing the involved concepts and their prototypical implementation, this section lists possible business implications that could be realized by running the ATP application in a productive environment.

Parallel ATP Checks on Hot Products. Performing ATP checks on aggregates necessitates exclusive locking. As demonstrated in the previous section, we can omit exclusive locks in our prototype. This increases the throughput of simultaneous checks on the same product by using multi-core technology. The limit of performing simultaneous checks on single products is a problem, e.g. for high-tech companies when introducing new, highly requested products.

Changing the Temporal Granularity for Every Check. Since for every check the corresponding time stamp is saved, it is possible to change the temporal granularity for every single check. Thus, a check on hours or on weeks can be done in the same system without any modifications. This is not possible with aggregates as they can only operate on one initially defined temporal granularity.

Considering Product Attributes in the Check. The inclusion of additional attributes during the check is supported by the columnar data structure and has a significant business impact. Now, companies are able to include fine-granular product attributes in their checks for otherwise identical products. Examples are the date of expiry in the food industry or the quality of raw materials e.g. in the steel industry.

Analytics on Check History. The ability to do analytics on the history of ATP checks introduces two major advantages: on the one hand, a company can perform classical analytical tasks such as seeing which products were sold the most, which customer groups ordered which kinds of products in which time periods and so on. On the other hand, storing the complete history of ATP checks also including those checks which did not result in actual orders, makes an important source of information accessible: companies can see which products were highly requested, but were not ordered e.g. because not enough quantities were available. Furthermore, a company can see which are the most popular replacement products, e.g. which products were ordered in the end, although different products were initially included in the ATP check.

Instant Order Rescheduling. Promised orders and planned deliveries are based on a certain schedule of incoming products or planned production. However, often these schedules and plans turn out to be incorrect as products may not arrive on time or production goals cannot be met. As a consequence, already promised orders have to be rescheduled. Using aggregates, this is a time-intensive process as the relevant

aggregates have to be calculated back to the point where the order in question was promised, so that the new delivery date can be calculated considering the changed stock projections. This operation can now be done significantly faster as all the relevant fine granular data is on hand. Including additional attributes such as prioritization of customers implies further functionality: A major customer's order that precedes other orders in its relevance can be protected from rescheduling activities incurred by unexpected changes to a product's stock level.

5 Conclusion

The real-time ATP approach presented in this paper does not only tackle performance bottlenecks, but also enables innovative features. On the technical side, we introduced the candidate checking algorithm and identified the instant reservation strategy as suitable concurrency control mechanism for executing an ATP check on an in-memory column-store. Based on a dataset with 64 million transactional records, we achieved a check time of 0.6 seconds. The dataset is based on the anonymized data from a Fortune 500 consumer products company and spans three years of operation. Our approach scales linearly with added CPU cores, even in hot-spot situations with checks against the same product. On the business side, we described possible business implications of our approach. To our knowledge, many ATP related systems do not track availability checks which did not result in orders and thereby lose extremely valuable information for planning purposes. Furthermore, rescheduling of orders is a time-intensive, static process without the possibility to include further requirements. Only these two aspects alone provide a significant benefit for companies.

The outlook of this paper leaves the safe harbor of well-established database concepts, prototypical implementations, and measurable results, but draws a vision of how companies could leverage analytical capabilities in the context of an ATP check in order to maximize their profits. On the one hand, let us assume that we can analyze the full history of a customer's ATP checks and resulting orders during every new incoming check. That implicates that we can calculate the probability of a customer still placing his order even if he cannot get the products delivered on his initially requested date. Therefore, we can derive a certain flexibility on the companies' side when to produce and ship an order without actually losing any orders. On the other hand, we heavily discussed the consideration of additional attributes during the check throughout the paper. Another example for such an attribute could be the varying production costs for the different products over time. Even if companies sell products for the same price over a certain period of time, the real production costs vary heavily due to changing raw material costs, different availability and costs of labor, and changing component suppliers. Putting those pieces together, instead of just considering the available quantities during the check, a company could also include varying production costs and therefore present an availability date that aims at maximizing the profits.

References

- [AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. ColumnStores vs. RowStores: How Different Are They Really? *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 967, 2008.
- [BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. *Very Large Data Bases*, 1999.
- [CGK01] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query Optimization In Compressed Database Systems. *International Conference on Management of Data*, 30(2), 2001.
- [Dic05] Jörg Thomas Dickersbach. *Supply Chain Management with APO*. Springer, Berlin, Heidelberg, 2005.
- [GS91] Goetz Graefe and Leonard D. Shapiro. Data Compression and Database Performance. pages 22–27, 1991.
- [Hen10] Doug Henschen. SAP Announces In-Memory Analysis Technology, 2010.
- [HLAM06] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance Tradeoffs in Read-Optimized Databases. *Very Large Data Bases*, 2006.
- [KGT⁺09] Jens Krüger, Martin Grund, Christian Tinnefeld, Jan Schaffner, Stephan Müller, and Alexander Zeier. Enterprise Data Management in Mixed Workload Environments. *2009 16th International Conference on Industrial Engineering and Engineering Management*, pages 726–730, October 2009.
- [KHK80] V. J. Kollias, M. Hatzopoulos, and J. G. Kollias. Database maintenance efficiency using differential files. *Information Systems*, 5(4):319–321, 1980.
- [KTG⁺10] Jens Krüger, Christian Tinnefeld, Martin Grund, Alexander Zeier, and Hasso Plattner. A case for online mixed workload processing. In Shivnath Babu and G. N. Paulley, editors, *DBTest*. ACM, 2010.
- [Mar04] Tim Martyn. Reconsidering Multi-Dimensional schemas. *ACM SIGMOD Record*, 33(1):83, March 2004.
- [Pla09] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. *Read*, 2009.
- [Sc05] Michael Stonebraker and Ugur Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, pages 2–11. IEEE Computer Society, 2005.
- [SL76] Dennis G. Severance and Guy M. Lohman. Differential files: their application to the maintenance of large databases. *ACM Transactions on Database Systems (TODS)*, 1(3):256–267, 1976.
- [SZ03] Rainer Scheckenbach and Alexander Zeier. *Collaborative SCM in Branchen*. Galileo Press GmbH, Bonn, 2003.

Cloud Storage: Wie viel Cloud Computing steckt dahinter?

Michael C. Jaeger und Uwe Hohenstein

Corporate Research and Technologies, Siemens AG
System Architecture & Platforms (CT T DE IT 1)
Otto-Hahn-Ring 6
D-81730 München
{uwe.hohenstein | michael.c.jaeger}@siemens.com

Kurzfassung: Durch die hohe Dynamik im Bereich des Cloud Computings entstehen neues Potenzial, aber auch Risiken bezüglich existierender und neuer Anwendungen. Eine wichtige Ressource bei der Nutzung von Cloud Computing ist sicherlich Cloud Storage. Unter Cloud Storage werden Datenspeicher subsumiert, die in unterschiedlichen Varianten von Cloud Computing Betreibern angeboten werden. Die wichtige Frage für die Nutzung von Cloud Storage ist hierbei, inwiefern auf technischer Ebene die wesentlichen Eigenschaften des Cloud Computings wie Elastizität, Skalierbarkeit und Kostenreduktion umgesetzt werden. Anhand von Beispielen wird deutlich, dass viele dieser Eigenschaften nach derzeitigem Stand nicht vollständig erfüllt werden.

1 Einleitung

Im Bereich Cloud Computing können wir eine große Veränderung im Markt feststellen, da etablierte und neue Anbieter mit neuen Produkten auftreten. Gleichzeitig sind große Investitionen zu beobachten, die in den Aufbau von Datenzentren und Plattformtechnologien fließen. Diese Veränderung schafft Potenzial in Form von neuartigen Anwendungen oder Kosteneinsparungen bei existierenden Anwendungen. Gleichzeitig entsteht die Gefahr, dass bisherige Geschäftsmodelle nicht mehr aufrecht erhalten werden können. Daher ist Cloud Computing für Unternehmen von großer Bedeutung.

Bausteine des Cloud Computings, wie zum Beispiel Virtualisierung oder Service-Orientierung, sind bereits zuvor in verschiedenen Formen auf technologischer Ebene realisiert worden. Jedoch waren bisher diese Bausteine nicht in dieser Breite und Kombination am Markt verfügbar und haben nicht zu den konkreten Vorteilen geführt, die einem das Cloud Computing nun bietet. Als Hauptvorteile zählen Armbrust et al. auf [AFG+09]:

1. Im Vergleich zu der Ressourcen-Kapazität, die durch eigene Mittel aufgebaut werden kann, scheinen die Ressourcen, die über Cloud Computing eingekauft werden können, praktisch unbegrenzt.

2. Die Investitionen sind sehr gering. Man geht praktisch keine relevanten Verpflichtungen ein, und Anschaffungskosten entfallen größtenteils. Zudem wird nach tatsächlichem oder reserviertem Verbrauch bezahlt. Dies ist gemeinhin als Pay-as-you-go-Prinzip bekannt.
3. Es ist möglich, punktuell Ressourcen je nach Bedarf mal mehr oder weniger zu beziehen und auch nur den aktuellen Bedarf zu bezahlen. Man spricht hierbei von der *Elastizität* der Ressourcen.

Betrachten wir die Entwicklungen in den vergangenen Jahren, sehen wir noch weitere Vorteile hinzukommen:

4. Bei einigen Angeboten ist der administrative Aufwand verglichen mit dem Betrieb einer eigenen Infrastruktur geringer.
5. Durch den Betrieb in Datenzentren wird ein gewisses Maß an Ausfallsicherheit und Hochverfügbarkeit erreicht, welches bei eigener Infrastruktur sonst nur durch erhöhte Investitionen erreicht würde. Z.B. ist das Aufsetzen und Administrieren von Datenbank-Clustern mit einem erheblichen Aufwand verbunden, der durch einen Cloud-Anbieter abgenommen werden kann.

Cloud Computing umfasst im Wesentlichen die Ressourcen Rechenkapazität, Applikationen und Speicher. In diesem Beitrag soll Letzteres im Vordergrund stehen. Bei Speicher beziehen wir uns dabei nicht auf Arbeitsspeicher, sondern auf persistenten Speicher, dem sogenannten *Cloud Storage*, den Anwendungen nutzen können.

Cloud Storage kann in zwei Formen genutzt werden: Zum einen ist es für Cloud Anwendungen sinnvoll, einen Speicher in der Cloud zu nutzen, anstatt einen eigenen Datenbankserver in die Cloud einzuspielen, aufzusetzen und zu administrieren. Zum anderen kann es auch für Anwendungen außerhalb der Cloud interessant sein, um z.B. Daten in einer Cloud auszulagern und dadurch Administrationskosten zu sparen. Da ein Cloud Storage direkt vom Internet aus erreichbar ist, wird zudem eine große Breite von Zugriffswegen ermöglicht.

Wenn Cloud Storage für neue Projekte in Betracht kommt, stellen sich unmittelbar folgende Fragen: Wie sind die Vorteile des Cloud Computings für Cloud Storage genau ausgestaltet? Was sind die technischen Begebenheiten? Ausgehend von existierenden Angeboten haben wir den Eindruck gewonnen, dass die gemeinen Vorteile derzeit nicht in allen Aspekten erfüllt werden. Im Folgenden diskutieren wir daher die Punkte Elastizität, Kosten und Skalierbarkeit exemplarisch an diversen Cloud Storage Lösungen. Zudem werden datenbankrelevante Punkte wie Performanz, Ausfallsicherheit und Administration beleuchtet. Damit soll eine Grundlage geschaffen werden, die Entscheidern dient, die Eigenschaften des Cloud Storage richtig einzuschätzen.

Der folgende Teil des Beitrags gliedert sich wie folgt: Im nächsten Abschnitt beschreiben wir die grundlegenden Konzepte der verschiedenen Cloud Storage Lösungen. Darauf folgend diskutieren und analysieren wir potenzielle Fallstricke und Probleme, illustriert durch Beispiele basierend auf dem momentanen Entwicklungsstand. Der Beitrag endet mit dem Fazit, das aus der Analyse gezogen wird.

2 Speicherkategorien in der Cloud

Cloud Storage ist eine bedeutende Basis-Ressource, die auf Infrastrukturebene im Rahmen des Cloud Computings angeboten wird. Mehrere Gründe sprechen dafür, eine Cloud Storage Lösung einzusetzen. Zum einen gelten die im vorangegangenen Abschnitt angeführten Vorteile und Motivationen für Cloud Computing. Zum anderen bieten Anbieter gleichzeitig nicht nur den Speicher, sondern führen auch Datensicherungen durch und bauen auf eine Mehr-Knoten-Architektur, der den klassischen Cluster Deployments nahekommmt [LMM09]. Dies bedeutet, dass Produkte angeboten werden, die über einen einfachen Speicher deutlich hinausgehen. Bezüglich der Speicherung von Daten in der Cloud gibt es grundsätzlich zwei architekturelle Anordnungen:

- Speicher für Datenbankanwendungen in der Cloud: Eine Anwendung läuft in der Cloud und benötigt eine Datenbank zur Verwaltung persistenter Daten. Auch wenn prinzipiell die Cloud-Applikation ein On-Premise-Datenbankserver nutzen kann, so ist es aus Sicherheitsgründen (der Zugriff auf das DBS erfolgt von außerhalb der eigenen Organisation) und aus Latenzgründen wenig praktikabel. Sofern keine geschäftlichen Gründe gegen eine Speicherung der Daten in der Cloud sprechen, ist ein Aufsetzen der Anwendung und des Cloud Storges in der Cloud die technisch optimale Lösung.
- Auch wenn die Anwendung nicht in der Cloud aufgestellt wird, kann man dennoch von Cloud Storage profitieren. Mit der Nutzung von Cloud Storage kann auf die Anschaffung wie auch die Administration eines eigenen Datenbankservers verzichtet werden. Insbesondere das Aufstellen einer hochverfügbaren Cluster-Lösung benötigt einen erheblichen Aufwand. Zudem erhält man durch den Anbieter eine weltweite Erreichbarkeit, die gleichzeitig eine hohe Anzahl von Nutzern bedienen kann. Weiterhin kann man von neuartigen NoSQL-Angeboten [NoSQL] von einfacheren Datenspeichern profitieren, die nicht die Komplexität für die Planung und Betrieb eines relationalen Datenbankservers erfordern.

Es gibt im Wesentlichen drei Kategorien von Cloud Storage, die im Folgenden diskutiert werden sollen: Blob Storage, Table Storage und echte Datenbankserver. Weitere Kategorien wie der Plattenspeicher (z.B. Amazon EBS oder Windows Azure Drives) oder vorgefertigte Images z.B. von Oracle für Amazon sollen hier außer Acht gelassen werden.

2.1 Blob Storage

Blob Storages sind gedacht für die Speicherung von großen Binär- oder Textdaten („binary large objects“) wie z.B. Bilder, Software oder XML-Dokumente. Das Konzept Blob ist bereits aus der Welt der relationalen Datenbankserver bekannt. Blobs lassen sich beispielsweise gut für Plattformen zur Speicherung und Verteilung von Dokumenten und Software nutzen. Beispiele für Blob Storage Angebote sind Amazons Simple Storage Service (S3) aus dem Amazon Web Service (AWS) oder der Blob-Service des Microsoft Azure SDK.

Die Grundkonzepte eines Blob-Storage sind *Container*, die einen eindeutigen Namen besitzen müssen und die eigentlichen *Blobs* enthalten. Ein Benutzer kann mehrere Container anlegen. Ein Blob besitzt ebenfalls einen Namen und besteht aus den eigentlichen Objektdaten, im Prinzip einer Datei entsprechend, und zusätzlichen Metadaten. Zu den vordefinierten Metadaten zählen HTTP-Metadaten (ETag, Last-Modified, Content-Length, Content-Type, Content-Encoding, Content-Language etc.), es lassen sich aber auch benutzerdefinierte Metadaten aufnehmen. Da der Objektname das Trennsymbol „/“ enthalten kann, lassen sich Hierarchien im Sinne einer Verzeichnisstruktur ausdrücken.

Typische Operationen für einen Blob Storage sind:

- Create / Delete *Container*
- Write / Read / Delete *BLOB-Objekt*
- List *BLOB-Objekte*
- Get / Set *Metadata / Properties*

Die Adressierung eines Containers oder Blobs erfolgt über einen speziellen Uniform Resource Identifier (URI):

- <http://photos.s3.amazonaws.com/2009/Barbados/beach.jpg>: Bei diesem AWS-Beispiel ist `photos` der Container, der mit einem vordefinierten `s3.amazonaws.com` zu versehen ist; das Blob ist `2009/Barbados/beach.jpg`. `2009` und `Barbados` können hier als Unterverzeichnisse des Containers aufgefasst werden.
- <http://myaccount.blob.core.windows.net/?comp=list&maxresults=10&include=metadata>: Auch hier ist der Teil `blob.core.windows.net` für Microsoft Azure vorgegeben. `myaccount` ist die Benutzerkennung. Mit der URI werden alle Container inklusive (`include=`) ihrer Metadaten aufgelistet (`comp=list`), wobei das Ergebnis auf maximal 10 Sätze beschränkt wird (`maxresults=`).

Der Zugriff erfolgt in der Regel über zwei Protokolle, SOAP und REST (Representational State Transfer (RFC 2616) [Fi00]) über HTTP(S). REST ist ein HTTP-basiertes Protokoll, das HTTP-Operationen wie GET, DELETE oder PUT für Datenmanipulationen benutzt. Da der Zugriff aus einer Programmiersprache heraus recht aufwändig ist, werden entsprechende Programmierschnittstellen (wie JetS3t für Amazon S3) für die gängigen Programmiersprachen angeboten, mit denen sich die Benutzung der komplizierteren HTTP-Programmier-APIs vermeiden lassen.

Neben diesen grundsätzlichen Funktionalitäten bieten Blob Services weitere Funktionen, um die Performanz beim Zugriff zu erhöhen. So unterscheidet der Microsoft Azure Blob Storage zwischen Block und Page Blobs, deren Zugriffs-API einmal für sequenziellen Zugriff und ein anderes Mal für quasi zufällig platzierte Zugriffe optimiert sind. Hier muss der Anwender bzw. die Anwendung beim Anlegen entscheiden, welcher Blob-Typ angelegt werden soll.

2.2 Table Storage

Ein Table Storage gehört zur Gruppe der kürzlich aufgekommenen NoSQL-Datenbanken [NoSQL] und dient der Speicherung von strukturierten Daten - in einer relativ unstrukturierten, großen Tabelle, im Folgenden *BigTable* genannt. Die Tabellenstruktur muss dabei nicht im Sinne einer CREATE TABLE Anweisung vorgegeben werden, sondern ergibt sich dynamisch aus den zu speichernden Datensätzen. Jeder Datensatz besteht aus einem lokal eindeutigen Identifikator und einer Menge von Attributname/-wert-Paaren. Die Tabellenstruktur passt sich dann dynamisch den aktuellen Daten an. Beispiele sind Google's Implementierung einer BigTable [CDG+06], Amazon SimpleDB und der Azure Table Service. Abbildung 1 zeigt eine typische BigTable, die sowohl Kleidungsstücke als auch Auto- und Motoradteile enthält.

ID	Category	Subcat	Name	Color	Size	Make	Model
01	Clothes	Sweater	Cathair Sweater	Pink	S, M, L		
02	Clothes	Pants	Designer Jeans	Blue, Yellow, Pink	30x32,32x32		
03	Car Parts	Engine	Turbos			Audi	S4
04	Motorcycle Parts	Bodywork	Fender Eliminator	Blue			

Abbildung 1: Beispiel eines BigTable

Für einen Table Storage ist charakteristisch, dass er keine vordefinierte Struktur besitzt. Zudem beziehen sich alle Operationen und Anfragen auf genau eine BigTable. Als unmittelbare Konsequenz sind keine Verbunde (Joins) zwischen Tabellen möglich. Mit Ausnahme von Bulk-Operationen stehen auch keine Transaktionen zur Verfügung; jede Einzeloperation ist atomar.

Der Zugriff erfolgt wie bei Blob Storages über SOAP und REST über HTTP(S) bzw. einfacher zu benutzende Programmierschnittstellen. Eine typische Anfrage als URI in Azure lautet

```
http://myaccount.table.core.windows.net/MyTable()?
$filter=(Model%20eq%20"S4")%20and%20(Color%20eq%20"Blue").
```

Hiermit werden aus MyTable alle Datensätze herausgefiltert, die der Bedingung Model="S4" AND Color="Blue" genügen. Azure bietet auch eine ADO.NET und eine LINQ-Schnittstelle (language-integrated queries), die aber nicht den vollen Funktionsumfang des Table Services bietet.

2.3 Datenbankserver

Die Rubrik der echten Datenbankserver stellt einen „virtuellen“ Datenbankserver für jeden Benutzer in der Cloud bereit. Der Begriff der Echtheit bezieht sich hierbei auf einen ähnlichen Funktionsumfang, wie man ihn von klassischen relationalen Datenbankservern erwarten würde. Der zugrunde liegende physische Datenbankserver kann prinzipiell mehrere Kunden bedienen, obwohl in der Regel jeder seinen eigenen Datenbankserver erhält, wobei aber auf einen physischen Rechner durchaus mehrere Datenbankserver laufen können. Beispiele sind Amazon RDS (Relational DB Service) mit einem MySQL-Server und Microsoft SQL Azure mit einem SQLServer.

Oracle bietet derzeit nur eine Lösung, ein Oracle-Image in der Amazon Cloud zu installieren¹. Letzte Variante erfordert dann aber weiterhin eine Speicherplatz-Provisionierung und eine nunmehr Fern-Administration eines klassischen Datenbankservers. Insofern werden wir derartige Lösungen nicht in Betracht ziehen.

Während Blob und Table Storages ein SOAP-Interface bieten und mit dem REST-Protokoll neue Wege gehen, um den Zugriff der Daten zu ermöglichen, bieten die Datenbankserver die üblichen APIs wie JDBC, ADO.NET etc. an, die mit einer speziellen Datenbank-URL, die einen generierten Servernamen beinhaltet, zu versorgen sind. Beispiele für SQL Azure und Amazon RDS sind:

- `sqlcmd -S t17j2515ow.database.windows.net
-U MyMasterUser@t17j2515ow -d MyDB`
`t17j2515ow` ist ein vom Betreiber vergebener Name für den Datenbankserver, `MyDB` ist der Name Datenbank.
- `mysql -h myinstance.crwjauvgijdf.us-east-1.rds.amazonaws.com
-P 3306 -u MyMasterUser -p`
`myinstance` ist der Name der Datenbank, `crwjauvgijdf` der vom Betreiber vergebene Datenbankserver-Name.

Im Wesentlichen stellen die Angebote eine fremdverwaltete Instanz eines Datenbankservers dar.

Für Applikationen, die in der Cloud laufen und einen Datenbankserver benötigen, sind diese Lösungen wesentlich leichter als klassische Datenbankserver zu nutzen. Der Zugriff aus einer Cloud-Applikation heraus auf einen lokal betriebenen Datenbankserver ist zum einen aus Sicherheitsgründen (aufgrund einer quasi obligatorischen Firewall in großen Unternehmen) in der Regel nicht möglich. Zum anderen wird wegen der schwer nachzubildenden Symmetrie der Skalierungseigenschaften eine derartige Architektur problematisch sein: Die Anwendung in der Cloud wird viel elastischer skalieren als der lokal betriebene Datenbankserver, so dass Letzterer sich zum Flaschenhals entwickeln wird.

Ein weiterer Vorteil ist, dass der Zugriff auch aus Applikationen außerhalb der Cloud möglich ist, d.h., der Cloud-Datenbankserver ist prinzipiell von überall verfügbar. Hier

¹ <http://aws.amazon.com/solutions/global-solution-providers/oracle/>

kann wie beim Table- oder Blob-Store der Vorteil der ubiquitären Erreichbarkeit ausschlaggebend sein. Weitere Vorteile sind die mögliche Nutzung einer größeren Bandbreite der Internet Anbindung des Providers, die eventuell lokal nicht zur Verfügung steht, oder die teilweise besseren nicht-funktionalen Eigenschaften, die beim lokalen Setup unter Umständen mit Aufwand zu erzielen sind.

3 Einschätzung der Kategorien bzgl. Der Cloud Computing Vorteile

Im Folgenden beleuchten wir die Cloud Storage-Lösungen bezogen auf die eingangs diskutierten allgemeinen Vorteile von Cloud Computing.

3.1 Elastizität hinsichtlich Datenvolumen

Beim *Blob Storage* ist die Elastizität größtenteils gegeben. Obwohl diverse Limitationen wie die Anzahl der Container je Kunde oder eine maximale Blob-Größe bestehen, wirken sich diese nicht negativ auf die Erweiterbarkeit aus, da bei den gängigen Anbietern quasi beliebig viele Blobs je Container möglich sind.

Auch beim *Table Storage* gibt es Limitationen, die aber eine Datenbankobergrenze durch die Anzahl möglicher Tabellen und einer maximalen Tabellengröße (z.B. 100 x 10 GB = 1 TB bei Amazon) definieren. Insbesondere die maximale Tabellengröße ist für die Elastizität des Datenvolumens kritisch, da ihr Erreichen kontrolliert und in Applikationen entsprechend durch Anlegen einer neuen BigTable oder eine Umverteilung reagiert werden muss. Hierfür empfiehlt Amazon beispielsweise, an eine Vorabpartitionierung zu denken, wodurch zusätzlich eine bessere Skalierung der Zugriffe erzielt wird. Eine entsprechende Unterstützung ist z.B. beim Azure Table Service bereits gegeben.

Bei den *Datenbankservern* ist die Elastizität derzeit nicht im erwarteten Bereich, d.h. nicht so umgesetzt, wie man es bei den allgemein kommunizierten Vorteilen des Cloud Computings erwarten würde: So ist die Datenbankgröße beim Anlegen einer Datenbank festzulegen. Teilweise relativ kleine Obergrenzen, z.B. zurzeit von 50 GB bei SQL Azure als Business Edition, tun ihr Übriges, wobei sich diese Limitationen in naher Zukunft – wie in der Vergangenheit auch – noch ändern können. So ist die Grenze bei SQL Azure innerhalb von wenigen Monaten von 10 GB auf 50 GB in der Business Edition angewachsen, provisionierbar in 10 GB-Schritten.² Zu beachten ist, dass eine festgelegte Datenbankgröße provisioniert wird. Reicht die Datenbankgröße nicht aus, so kann eine neue Obergrenze über Administrationskommandos bis zur Maximalgröße vereinbart werden. Zurzeit sind unbegrenzt anforderbar die Anzahl der Datenbankserver und die Anzahl der Blobs; die Anzahl der Tabellen ist in der Regel limitiert.

² Angaben für Microsoft Azure abgerufen aus dem Internet unter <http://www.microsoft.com/en-us/sqlazure/offers/default.aspx> vom Sept. 2010

Eine generelle Einschränkung ist, dass besonders große Anzahl Instanzen (Datenbankserver, BigTables etc.) einer gesonderten Beantragung bedürfen und in der Regel nicht über eine Anbieter-Web-Anwendung automatisiert provisioniert werden. Letzten Endes tragen diese Mechanismen, ebenso wie einzelne voreinstellbare Provisionierungsfenster, zur Sicherheit des Kunden bei der Nutzung bei: Auf diese Weise soll verhindert werden, dass durch Irrtümer oder Programmierfehler ein zu großes Kontingent (automatisiert) angefordert wird, wodurch sonst hohe Kosten entstehen können.

3.2 Ausfallsicherheit und Hochverfügbarkeit

Die *Blob* und *Table Storages* replizieren die Daten in der Regel dreifach auf verschiedene Standorte. Zudem passiert ein Failover bei Ausfall eines der Speicher automatisch auf einen Anderen. Auch die *Datenbankserver* werden früher oder später automatisch replizieren, auch wenn es momentan noch nicht immer Standard ist, z.B. zurzeit bei Amazon RDS. Bei RDS kommt auch noch ein 4-stündiges Wartungsfenster je Woche hinzu, das festzulegen ist und in dem es z.B. bei Datenbank-Patchinstallationen durchaus zu einem Betriebsausfall kommen kann. Bedeutend ist, dass die Cloud-Datenbankserver „managed“ Services sind, d.h., das Einspielen von Patches erfolgt automatisch, wodurch die Administration vereinfacht wird.

Ein kompletter Datenverlust ist nicht zu befürchten, da Backup/Restore-Möglichkeiten existieren. Zum Beispiel spezifiziert Amazon für den S3 Blob Storage eine Haltbarkeit („Durability“) von „9-11“³, was einen Verlust von einem aus 10.000 Blobs alle 10 Millionen Jahren gleichzusetzen ist. Gleichzeitig gibt es für die Kunden die Möglichkeit eine 9-4 Haltbarkeit für einen Datenbereich festzulegen (Reduced Redundancy Storage, RSS), welcher preiswerter angeboten wird.

Bezüglich der allgemeinen Verfügbarkeit haben Microsoft als auch Amazon gleiche Angebote: Grundsätzlich wird eine 99.9% Verfügbarkeit garantiert, die bei Verletzung in Form eines 10% Rabatts kompensiert wird. Sinkt die Verfügbarkeit unter 99%, wird durch beide Anbieter zurzeit ein Rabatt von 25% gewährt.⁴ Hierbei ist zu beachten, dass nur auf gezahlte Beträge ein Rabatt eingeräumt wird. Es findet also kein Transfer von Beträgen vom Anbieter zum Kunden statt. Die Verfügbarkeit wird zudem durch Begrenzung der Dauer einer Abfrageausführung limitiert; wird ein Schwellwert überschritten, wird die Abfrageausführung abgebrochen. Zudem können erneute Anfragen verzögert werden, um Denial-of-Service-Angriffe entgegen zu wirken. Durch diesen Mechanismus kann schnell die durch einen (missglückten) Test erzeugte Last während der Entwicklung zu einer Sperrung der Datenbank führen, was während der Entwicklung zu beachten ist.

³ 9-11 bedeutet eine Verfügbarkeit von 99.x % mit 2 Stellen vor und 9 Stellen hinter dem Komma.

⁴ Angaben für Amazon abgerufen aus dem Internet unter <http://aws.amazon.com/simplydb/> vom Sept. 2010

Eine Änderung der Instanzgröße ist jederzeit möglich, wobei eine Ressourcenerweiterung einen möglichen Betriebsausfall zur Folge hat. Ein Wechsel der Instanzgröße wird häufig den Umzug des Datenbankservers oder der Datenbank auf anderen Rechner bedeuten. In der Regel wird ein Wechsel der Instanzgrößen aber durch eine Hot-Switch/Staging Technik gelöst; es wird zunächst parallel eine neue Instanz gestartet, auf die umgeschaltet wird, um die Originale abzulösen.

3.3 Kosteneinsparung durch Elastizität

Die Kostenmodelle unterliegen derzeit noch starken Schwankungen und werden sich vermutlich auf einheitlichem Niveau einpegeln. Die Kosten, die einen erwarten, sind schwer überschaubar, da es neben den reinen Speicherkosten sehr viele Einflussfaktoren wie z.B. die CPU-Benutzung, Datentransferkosten von der (das Ergebnis) und in die Cloud (die Anfrage) gibt und mitunter XML-Daten transportiert werden, deren Größe nicht einfach abschätzbar ist. Zudem gibt es unterschiedliche Preise für die jeweils angebotenen Regionen (z.B. US East/West, EU)

Bei den *Blob Storages* treten bereits recht hohe Speicherkosten von 150\$ für 1 TB pro Monat bei Amazon S3 auf, während eine eigene TB-Festplatte durchaus bereits für 100\$ zu haben ist. Ein fairer Vergleich muss allerdings auch die Replikation und die eingesparte Administration (Restarts, Updates, Patches, etc.) berücksichtigen. Hinzu kommen noch die Transferkosten. Die Kosten skalieren mit dem Datenvolumen und den anderen Faktoren, wobei die relativen Kosten mit wachsendem Verbrauch abnehmen, z.B. von anfangs 15Ct für 1 GB auf 5,5Ct bei einer Belegung von über 5000 TB.

Bei den *Table Storages* sind der Speicherplatz und weiterer Ressourcenverbrauch häufig bis zu einer gewissen Größe frei, erst danach fallen Kosten an. Das Kostenmodell ist insofern für Privatpersonen und Start-ups interessant, als bei kleinen Datenmengen (häufig kleiner als 1GB) erst einmal keine Kosten anfallen, weder Fixkosten noch Verbrauchskosten. Die Nutzungskosten orientieren sich wieder an vielen Faktoren und wachsen mit dem Datenvolumen und den Zugriffen.

Andere Kostenmodelle gibt es für die *Datenbankserver*: Hier fallen Kosten bereits beim Anlegen einer Datenbank an, wobei in der Regel die Datenbankkonfiguration (Small, Large, bis hin zu Quadruple Extra Large bei Amazon RDS), die Speicherprovisionierung (anzugeben beim Anlegen der Datenbank), also *nicht* der aktuell belegte Speicher, sowie der Datentransfer einfließen. Der festgelegte Benutzungsrahmen ist zu bezahlen. Auch ist das Volumen von Backup-Daten (z.B. durch eine angegebene Backup-Periode gesteuert) zu berücksichtigen.

Bei SQL Azure ist das Preismodell recht einfach, da hier ein fester Preis für die aktuelle Datenbankgröße bis zum nächsten Provisionierungsschritt zu bezahlen ist. Dies bedeutet bei den zurzeit eingerichteten 10GB Schritten in der Business Edition, dass eine 13GB Datenbank zum Preis von 20GB abgerechnet wird, auch wenn die provisionierte Maximalgröße („cap“) 50GB beträgt. Bei der von [KKL10] durchgeführten Untersuchung führte dieser Punkt zu den günstigen Kosten. Auch bei den Cloud-Datenbankservern muss ein fairer Preisvergleich mit selbstadministrierten

Datenbankservern die bereitgestellte Ausfallsicherheit wie auch eingesparten Administrationskosten berücksichtigen, da gerade das Aufsetzen und Administrieren eines Datenbank-Clusters eine aufwändige Angelegenheit ist.

3.4 Administration

Beim *Blob Storage* fallen keine eigenen Administrationskosten an, sieht man einmal von der Registrierung und dem Anlegen einer Benutzerkennung ab.

Auch bei den *Table Storages* entstehen auf dem ersten Blick keine Administrationskosten. Typische DBA-Aufgaben werden vom Anbieter übernommen und können auch nicht selber wahrgenommen werden, z.B. übernimmt der Azure Table Service eine automatische Indexierung der BigTables, wobei sich sofort die Frage stellt, ob der Automatismus wirklich gut genug für eine optimale Performanz ist. Problematisch ist die Größenbeschränkung auf Tabellenebene, die eine Überwachung der Tabellengröße erfordert wie auch dann eine adäquate Reaktion beim Erreichen des Maximums.

Bei den *Datenbankservern* wird die Hardwareausstattung der Datenbankserver (Anzahl CPUs, Anzahl Core's, Platten etc.) gemäß der gewählten Instanzgröße bereitgestellt. Eine Administration entfällt also weitgehend bzw. ist gar nicht möglich. Eine manuelle Überwachung wird auch hier nötig, um bei Erreichen der Maximalgrößen reagieren zu können. Beispielsweise kann bei einer auftretenden Exception aufgrund einer erreichten Quotagrenze ein Administrationkommando abgesetzt werden, durch das die Maximalgrenze verschoben wird. Ebenso ist die Performanz zu kontrollieren, um ggf. in eine andere Instanzkategorie zu wechseln. Performanzkritische DBA-Aktionen wie das Erzeugen von Indexen haben weiterhin Bestand. Als Fazit lässt sich feststellen, dass die DBA-Kosten nicht entfallen. Insbesondere mangelt es an einer automatischen Kontrolle der Speicherbelegung bzw. automatischen Erweiterung bei Erreichen der Maximalgröße.

4. Allgemeine Punkte

4.1 Skalierbarkeit

Bei *Blob* und *Table Storages* gehen die Requests über das REST- oder SOAP-Protokoll an einen Web-Server, d.h., die Skalierung der Zugriffe liegt als Erstes in der Verantwortung des Web-Servers und wird von dort aus auf das Speichersystem verteilt. Eine implizite Replikation der Daten auf in der Regel drei physikalische Standorte trägt zudem zur Skalierbarkeit bei. Der Server selbst ist nicht zu beeinflussen.

Ein selbstverwalteter *Datenbankserver* wird ohne Umweg über einen Web-Server angesprochen. Insofern ist die Skalierbarkeit des Datenbankservers ausschlaggebend. Der Datenbankserver lässt sich in der Regel bzgl. des internen Caches, der Anzahl erlaubter Transaktionen, der benutzten Platten etc. konfigurieren bzw. mit parallelen Platten oder RAID-Systemen zu versorgen, um einen optimalen Betrieb und eine hohe

Skalierbarkeit zu gewährleisten. Amazon RDS und Microsoft SQL Azure sehen keine derartige Konfiguration vor, sondern bieten nur vorgefertigte Instanzkategorien wie Small, Large, XL etc. an, die beim Anlegen der Datenbank auszuwählen sind. Hinter jeder Kategorie verbirgt sich eine entsprechende Rechnerausstattung, z.B. 2 CPU-Kerne, 3.5 GB Hauptspeicher und 500 GB Plattenspeicher bei der SQL Azure Medium-Konfiguration. Die Skalierbarkeit erfolgt somit nur über die Datenbankserver-Konfiguration, insbesondere der maximale Anzahl erlaubter Verbindungen der jeweiligen Instanzgröße. Grundsätzlich ist keine weitergehende Einflussnahme auf die Hardware wie das Hinzuschalten weiterer Platten oder Disk Arrays möglich.

Eine nachträgliche Änderung ist als Reaktion auf beobachtete Skalierungs- oder Performanzprobleme möglich, aber auch hier stellt sich die Frage sowohl nach der Zeit zwischen Anforderung und Umsetzung bei Ressourcenerweiterung als auch einem möglichen Betriebsausfall.

4.2 Performanz

Ein signifikanter Unterschied zum klassischen Datenbankserver dürfte durch den Zugriff zum Cloud Storage an sich entstehen. Ein Cloud Storage ist im Internet verfügbar und ein darauf gerichteter Zugriff geht den bekannten Weg über Router und Internet-Anbieter. Hierbei sind deutliche Unterschiede in Latenz und Durchsatz im Vergleich zum lokal aufgestellten Datenbankserver zu erwarten. An dieser Stelle muss die Art der Anwendung entscheiden, ob Einschränkungen bzgl. der Latenz und der Bandbreite beim Zugriff in die Cloud einen Einsatz erlauben. Dieser Nachteil ist nicht gültig, wenn die Anwendung, die auf den Cloud Storage zugreift, selbst in der Cloud aufgestellt wird.

Üblicherweise hat der Datenbankadministrator mit dem Anlegen von Indexen einen direkten Einfluss auf die Performanz beim Datenzugriff auf ein Datenbanksystem. Die Implementierungen der Table Storages der unterschiedlichen Anbieter sehen im Gegensatz dazu vor, dass der Anwender nicht „mit der Erstellung von Indexen belastet wird“, wie es Amazon in der entsprechenden Dokumentation formuliert. Auch beim Azure Table Service wird nur das Vorhandensein fest eingeplanter Indexe dokumentiert, aber keine Möglichkeit geboten, eigene Indexe anzulegen. Dies bedeutet nicht, dass die Table Storages im Zugriff per se langsamer sind; es wird lediglich dem Benutzer die Möglichkeit genommen, direkt Einfluss auf die Datenorganisation zu nehmen. Abhilfe kann hier eine frühzeitig festgelegte Partitionierung schaffen, die bei Azure recht einfach über einen Partition Key steuerbar ist.

Weiterhin ist zu beachten, dass die Anbieter von Cloud Storages ein sogenanntes Throttling einsetzen um eine unausgewogene Nutzung der Ressourcen zu vermeiden. Im Extremfall dient Throttling dazu, eine Denial-of-Service-Attacke zu verhindern, im Normalfall wird durch Throttling eine Abgrenzung zwischen verschiedenen Produktklassen bzw. Service Level Agreements (SLA) hergestellt. Beispielsweise ist bei einer kostenlosen Nutzung der Google App Engine der Datendurchsatz auf knapp ein 1MB/sec beschränkt, wohingegen im Bezahlmodus der maximale Datendurchsatz über 150MB/sec betragen kann. Dies bedeutet, dass bei einer intensiven Nutzung von Cloud Storage die Throttling Einstellungen des Anbieters relevant sind.

Bei den Datenbankservern wird die Performanz, wie bereits erwähnt, indirekt über die Instanzgröße gesteuert. Bei Problemen mit der Verarbeitungsgeschwindigkeit kann nur durch einen Wechsel der Instanzkategorie die Anzahl der CPUs etc. erhöht werden. Zu beachten ist auch, dass man nur einen *virtuellen* Datenbankserver erhält. Es ist also nicht gewährleistet, dass einem ein *physischer* Datenbankserver allein zur Verfügung stehen wird; prinzipiell kann der Datenbankserver gleichzeitig mehreren Kunden bzw. virtuellen Instanzen zur Verfügung stehen. Häufig wird ein physikalischer Rechner mehrere Datenbankserver, bis zu vier bei SQL Azure, aufnehmen; eine gegenseitige Beeinflussung scheint vorprogrammiert und hat bei klassischen Datenbankservern massiv zu Performanzproblemen geführt. Aus diesem Grund gibt es vermutlich nur die Zusicherung der Verfügbarkeit per SLA, aber kein zugesagtes Antwortverhalten.

4.3 Standardisierung

Wie bereits erläutert, ist ein Table Storage nicht mit einem klassischen Datenbankserver zu vergleichen [Sh07], da sich alle Operationen, auch lesende Anfragen, auf genau eine BigTable beziehen. Beziehungen zwischen Tabellen sind folglich nicht als Join in der Abfragesprache herzustellen. Die Gründe hierfür sind zweierlei: Zum einen wird durch diese Vereinfachung eine optimale Skalierbarkeit und Partitionierung der Daten und damit eine optimale Performanz ermöglicht. Zum anderen orientieren sich die Table Storages an einem Speicher für Objekte, so dass Abfragesprachen unabhängig vom relationalen Modell gestaltet sind. Dies wird durch eine Übersicht der Abfragesprachen deutlich:

Google App Engine:	JDO Query Language (JDOQL)
Microsoft Azure Cloud:	Operatoren und Ausrücke via REST oder LINQ
Amazon SimpleDB:	SELECT Ausdruck mit Filter und Sortierung, keine JOIN Ausdrücke

Die Beschaffenheit der Abfragesprachen zeigt, dass bei der Benutzung eines Table Storages Beziehungen zwischen Datenobjekten auf der Ebene der Anwendungslogik aufgelöst werden müssen. Weiterhin wird deutlich, dass ein Äquivalent zum weit verbreiteten SQL bei Table Storages noch nicht vorhanden ist. Gleiches gilt auch, wenn man beispielsweise die Syntax einer via REST formulierten Abfrage an SimpleDB mit einer Anfrage an den Azure Table Storage vergleicht. Bei SimpleDB wird ein SELECT-Ausdruck als Parameter einer REST-Anfrage verpackt. Bei Microsoft Azure wird der Ausdruck auf die Schlüssel-Wert-Paare einer REST-Anfrage abgebildet. Hierbei muss allerdings angemerkt werden, dass in den meisten Fällen ein clientseitiger Proxy den Zusammenbau der REST-Anfrage erledigt. Dennoch wäre für die Vereinheitlichung von unterschiedlichen Client-Proxies eine Standardisierung für REST-Aufrufe für die unterschiedlichen Table Storage-Produkte von Vorteil.

Weiterhin ist eine Vereinheitlichung der Schnittstelle in einer Hochsprache für Table Storages nicht so zu finden, wie es mit klassischen relationalen Datenbankservern mit JDBC oder ODBC der Fall ist. Innerhalb der Java-Welt ließe sich JDO als objektorientiertes Mapping zum Table Storage vorstellen. Ein Standard wie JPA, also eine API, die sich nur auf Table Storages beschränkt, ist allerdings noch nicht

vorhanden. Gleichwohl gibt es erste Ansätze wie SimpleJPA [SJPA] oder SimpleORM [SORM] für einzelne Produkte, die in diese Richtung gehen.

Bei den Storage Produkten, die einen SQL Server zur Verfügung stellen, handelt es sich um Weiterentwicklungen auf Basis bereits bekannter Produkte (Microsoft SQL Server bzw. MySQL bei Amazon RDS). In diesem Fall sind die üblichen Standards vorhanden und ähnliche Feinheiten bezüglich Einrichten, Verhalten und Funktionalität zu erwarten, wie sie auch die bisherigen Produkte aufgezeigt haben.

4.4 Weitere Besonderheiten

Für die Migration bestehender Applikationen ist es bedeutend, dass es bei Amazon SimpleDB und bei Amazon S3 nur einen Datentyp gibt (nur Strings im UTF-8 encoding) und dass alle Attributwerte kleiner als 1KB sein müssen. Die Konsequenzen können dann bei einer Migration von einem klassischen Datenbankserver zu Amazon SimpleDB erheblich ausfallen, wenn Anwendungslogik, die auf Wertevergleich von Zahlen basiert oder Blobs verarbeitet, auf die Verarbeitung von diesen Strings geändert werden muss.

Google's BigTable erlaubt die Speicherung auf Basis von JSON-Definitionen (JavaScript Object Notation). Dies ist ebenso das Format, welches von den Softwareprodukten CouchOne oder MongoDB über deren APIs verarbeitet wird. Da JSON nicht nur Strings, sondern auch Zahlen oder boolesche Werte kennt, ist hier die Ähnlichkeit zu einem klassischen Datenbankserver viel größer als bei SimpleDB von Amazon. Der Table Service aus der Microsoft Welt verarbeitet die Datentypen der ADO.NET Data Services Spezifikation.

Bei SQL Azure sind wenige nennenswerte Unterschiede zu dem bisherigen SQL Server Produkt zu nennen, die nach und nach behoben werden. Erst kürzlich hat Microsoft die Unterstützung von hierarchischen Daten in SQL Azure als Datentyp nachgezogen und eine Unterstützung für Datenverschlüsselung angekündigt. Einige Datentypen früherer SQL Server Versionen werden wie von der aktuellen Version auch nicht von SQL Azure unterstützt (text, image, ntext). Da der Amazon RDS im Prinzip eine Version von MySQL darstellt (zurzeit MySQL 5.1.50), ist ein Transfer der Daten von einem MySQL Server zu Amazon RDS problemlos möglich.

Neben den genannten Punkten muss noch beachtet werden, dass die Testbarkeit durch den Einsatz von Cloud Storage leidet und im Normalfall Kosten verursacht. An dieser Stelle bieten die Anbieter teilweise Arbeitsumgebungen mit lokaler Datenbank, um diesen Nachteil zu kompensieren, zum Beispiel Microsoft mit der Developer AppFabric.

Schließlich können beim praktischen Einsatz der Cloud Storage Lösungen noch unerwartete Überraschungen auftreten: So kann es beim Zugang aus dem Firmennetz auf einen Cloud-Datenbankserver Probleme mit dem eigenen Proxy geben, der keine Requests über spezielle Ports hinauslässt. Diese können zum Beispiel von einem Administrationswerkzeug für SQL Azure oder Amazon RDS stammen. Der Zugriff auf Blob und Table Storages mit SOAP oder REST ist hingegen kein Problem.

5 Zusammenfassung und Ausblick

In dieser Übersicht wurde aufgezeigt, dass Eigenschaften, die man seitens klassischer Datenbankserver kennt, nicht per se im Bereich der Cloud Storages vorhanden sind. Beispiele hierzu lassen sich auch in [Sh07] nachlesen. Die Eigenschaften unterscheiden sich hinsichtlich der Produktabgrenzungen und hinsichtlich der Anbieter. Standardisierungen in der Benutzung und der Eigenschaften bleiben insbesondere für die neue Technologie der NoSQL-Datenbanken noch aus. Für den Anwender bedeutet dies, dass Kostenmodelle und Performanz der unterschiedlichen Produkte gegenübergestellt werden müssen, um eine Entscheidung zur Nutzung eines einzelnen Produktes treffen zu können.

Auffällig ist auch, dass bei technischer Betrachtung der Möglichkeiten die grundlegenden Vorteile von Cloud Computing, Elastizität, Skalierbarkeit und die Möglichkeit der Kostensenkung im Einzelfall unter Umständen nicht erreicht werden:

- Die Skalierbarkeit ist wie auch die Elastizität durch den Nutzungsrahmen (z.B. die maximale Datenbank- oder Tabellengröße, oder die Anzahl zur Verfügung stehender Datenbankverbindungen) der konkreten Produkte beschränkt. Bei einigen Nutzungsmodellen gibt es klare Obergrenzen, deren Überschreitung eines administrativen Aktes bedarf, sofern man sich nicht frühzeitig aus Vorsicht auf einen zu hohen und damit kostspieligeren Nutzungsrahmen festlegen möchte.
- Die Kosten können bei häufigen Datentransfer zur und von der Cloud u.U. größer ausfallen als bei Applikationen, die in der Cloud laufen. Hierbei können Server mit großen lokalen Festplatten wiederum eine preiswerte Alternative darstellen. Zwar bietet Amazon mit dem Reduced Redundancy Storage eine günstige Alternative an, die wiederum hinsichtlich der Zuverlässigkeit stark eingeschränkt ist. Letzten Endes entscheidet die Art Anwendung über den optimalen Einsatz von Storage Lösungen.
- Die Ausfallsicherheit ist nur in dem Umfang der Produktausgestaltung gegeben. Unter Umständen sind Wartungsfenster zu beachten, die die Verfügbarkeit im Vergleich zu lokal betriebenen Lösungen einschränken.
- Der administrative Aufwand kann in der Tat geringer ausfallen, allerdings auch unter dem Verlust der Kontroll- und Gestaltungsmöglichkeit. Hier müssen die Anforderungen der Anwendungen bzw. die erzielte Performanz entscheiden, ob die Einschränkungen nicht zu Nachteilen führen.

Dies Punkte zeigen deutlich, dass in vielen Bereichen noch Nachbesserungen erforderlich sind: Die in der Breite suggerierten Eigenschaften des Cloud Computing sind technisch bei den momentanen Cloud Storage Angeboten nicht konsequent umgesetzt.

5.1 Ausblick

Die genannten Arten von Cloud Storage und die verfügbare Produktvielfalt von Blob über Table Storages hin zu Datenbankservern machen insgesamt deutlich, dass viele

Faktoren die Auswahl eines Produktes beeinflussen. Betrachtet man dazu die Freiheitsgrade bezüglich der Aufstellung der eigenen Anwendung, wird klar, dass die Nutzung von Cloud Storage zu unterschiedlichen Formen der Anwendungsarchitektur führen kann. Insbesondere eine Partitionierung der Daten sollte aufgrund existierender Obergrenzen und Skalierungsgründen frühzeitig berücksichtigt werden. Mangelnde Erfahrungen mit Frameworks wie Hibernate oder das Java Persistence API (JPA) machen sich momentan noch negativ bemerkbar. Zudem wird insbesondere der Kostenfaktor die technologische Auswahl wie auch die Architektur beeinflussen. Auch gesetzliche Aspekte sind sicherlich relevant, wie z.B. wo die Daten letztendlich liegen und welchen lokalen Gesetzgebungen (z.B. der PATRIOT Act der U.S.A.) sie unterliegen.

Ein wichtiges Thema ist weiterhin die Migration bestehender Datenbanken und/oder Applikationen in die Cloud. Hier stellt sich heraus, dass Table Storages mit einer Portabilität von Anwendungen schlecht verträglich sind. Zum einen bedeutet die Migration einer bestehenden relationalen Datenbank in einen Table Storage einen erheblichen Aufwand, nicht nur bei der Datenkonvertierung sondern auch bei den Zugriffen. Zum anderen wird ein späterer Wechsel des Betreibers nicht einfach werden (Datentypen von Amazon SimpleDB gegenüber denen vom Azure Table Service), und insbesondere wird der Rückweg zu einem relationalen Datenbankserver erschwert. Die Migration zwischen klassischen Datenbankservern und Cloud-Datenbankservern gestaltet sich aufgrund ihrer relativen Verwandtschaft einfacher, auch wenn ihr Betrieb in der Cloud Einschränkungen unterliegt. Nichtsdestoweniger bleiben die APIs gleich.

Und schließlich blieben die administrativen Probleme in Unternehmen bisher unerwähnt: Als Privatperson ist es recht einfach die Kreditkarte anzugeben, die dann mit den Verbrauchskosten belastet wird. In vielen Firmen muss allerdings vor der Nutzung ein Bestellvorgang initiiert werden, der einen Preis benennt. Wird dieser Preis dann aufgrund des Pay-as-you-go überschritten, so können die zusätzlichen Kosten nicht mehr abgerechnet werden. Hier sind sicherlich noch Nachbesserungen in den Tarifoptionen notwendig, um den breiten Einsatz in Unternehmen zu erleichtern.

Literaturverzeichnis

- [AFG+09] Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia: Above the Clouds: A Berkeley View of Cloud Computing. UC Berkeley Reliable Adaptive Distributed Systems Laboratory, Technical Report EECS-2009-28, February 2009 available online at <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html>
- [CDG+06] F. Chang, J. Dean, S. Ghemawat, et al.: Bigtable: A Distributed Storage System for Structured Data. In OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, 2006
- [CH09] D. Chappell: Introducing Windows Azure. White paper, Microsoft and Chappell Associates, December 2009. Available online (25 pages).
- [Fi00] R. T. Fielding: REST: Architectural Styles and the Design of Networkbased Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.
- [KKL10] D. Kossmann, T. Kraska, S. Loesing: An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. ACM SIGMOD 2010

- [LMM09] J. Lee, G. Malcolm, and A. Matthews: Overview of Microsoft SQL Azure Database, Microsoft Technical Whitepaper 2009.
- [Mongo] Mongoddb - scalable, high-performance, open source, document-oriented database. Available from <http://www.mongodb.org/>.
- [MS1] Microsoft SQL Azure Database <http://www.microsoft.com/azure/data.mspx>
- [NoSQL] <http://nosql-database.org>
- [Sh07] N. Shalom: Amazon SimpleDB is not a database! http://natishalom.typepad.com/nati_shaloms_blog/2007/12/amazon-simpliedb.html
- [SJPA] SimpleJPA, version 1.0 project homepage, accessed Feb 2010 at <http://code.google.com/p/simplejpa/>
- [SORM] SimpleORM, version 3.* project homepage, accessed Feb 2010 at <http://www.simpleorm.org/>

Panel: “One Size Fits All”: An Idea Whose Time Has Come and Gone?

Jens Dittrich, Franz Färber, Goetz Graefe, Henrik Loeser, Wilfried Reimann

Moderation: Harald Schöning

Mit seiner Publikation “One Size Fits All: An Idea Whose Time Has Come and Gone?” [St05] hat Michael Stonebraker 2005 die Datenbankszene provoziert. Er vertritt dort die These, dass die eingeführten und in vielen Kontexten eingesetzten „traditionellen“ Datenbanksysteme zwar für viele datenzentrische Anwendungen eingesetzt werden, aber keineswegs die optimalen Systeme für viele dieser Anwendungen (OLAP, Stream Processing, XML Datenbanken, wissenschaftliche Datenbanken, Textsuche) sind. In einer späteren Publikation [St07] stellte er sogar die These auf, dass es keine Anwendungen gibt, für die die traditionellen Datenbanksysteme die beste Alternative sind. Er sagte voraus, dass sich in der kommerziellen Welt eine Anzahl unabhängiger spezialisierter „neuer“ Datenbanksysteme aufsplitten werde, die bestenfalls durch ein gemeinsames Frontend verbunden sind.

Gut fünf Jahre später wollen wir nun diskutieren, inwieweit Stonebrakers Thesen sich bewahrheitet haben und wie sie aus heutiger Sicht zu bewerten sind. Fünf Experten werden dazu aus unterschiedlichen Perspektiven Stellung beziehen:

Jens Dittrich ist seit 2008 Professor für Informationssysteme an der Universität des Saarlandes. Nach seiner Promotion an der Universität Marburg hat er von 2003 bis 2004 für die SAP AG gearbeitet im Bereich OLAP/TREX/BI Accelerator. 2004 wechselte er als Postdoc zur Systems Group an die ETH Zürich. Schwerpunkte seiner Forschung sind effiziente Verfahren für sehr große Datenmengen. Seine derzeitigen Projekte sind Hadoop++ sowie OctopusDB. Seine Arbeit “Towards a One Size Fits All Database Architecture” wurde 2011 auf der Conference on Innovative Data Systems Research (CIDR) als “Best Outrageous Ideas and Vision Track Paper” ausgezeichnet.

Franz Färber ist Chief Software Architect im TREX Team der SAP. Das TREX Team zeichnet für den Entwurf der Column-Store Engine im SAP NetWeaver Business Warehouse Accelerator verantwortlich und entwickelt dieses System. Vor seinem Wechsel zu SAP im Jahre 1994 arbeitete Franz Färber als Entwickler bei IBM: Seine Schwerpunkte sind Datenstrukturen, Kompression und Datenzugriffsmethoden.

Goetz Graefe arbeitet seit vier Jahren in den Hewlett Packard Laboratories und ist seit 2008 HP Fellow. Davor war er 12 Jahre für Microsoft tätig und arbeitete dort im Bereich SQL Server vor allem an Anfrageoptimierung und effizienter Anfrageverarbeitung. Zahlreiche Auszeichnungen, darunter der “ten-year test of time award” der ACM Special Interest Group on Management of Data (SIGMOD) International Conference und der “influential paper award” der IEEE International Conference on Data Engineering (ICDE) belegen, dass er ein anerkannter Experte auf seinem Gebiet ist.

Henrik Loeser arbeitet als Architekt und Evangelist im Information Management Technology Ecosystem (IMTE) Team der IBM. In den letzten Jahren hat er sich als Architekt mit pureXML im Datenbanksystem DB2 sowie mit dem IBM Smart Analytics Optimizer auseinandergesetzt. Zu seinen Zeiten an der Universität Kaiserslautern waren Non-Standard, objekt-orientierte und objekt-relationale Datenbanksysteme seinen Schwerpunkten. Als Hobby arbeitet er als Dozent für Datenmanagement an der Dualen Hochschule Baden-Württemberg Ravensburg.

Wilfried Reimann ist IT Senior Manager bei der Daimler AG. Er ist verantwortlich für Enterprise Architecture Integration und Innovation-Management. In den 30 Jahren seiner Tätigkeit bei Daimler hat er Expertise in den Bereichen Software-Architektur großer Unternehmenssoftwaresysteme, objektorientierte Technologie und Enterprise Architecture Management erworben.

Moderation: **Harald Schöning**, Senior Director in der Research-Abteilung der Software AG. Harald Schöning war vorher als ehemaliger Entwickler und Architekt von Adabas, Tamino (einer XML Datenbank) und verschiedener Produkte aus „neueren“ Technologiebereichen (semantische Informationsintegration, Complex Event Processing) tätig. Aus dieser Tätigkeit und seinen Lehrtätigkeiten an verschiedenen Universitäten resultiert das Interesse am Thema des Panels, das sich auch durch die von ihm geschriebenen Bücher zeigt.

Literaturverzeichnis

- [St05] Stonebraker, M., Cetintemel, U.. “One Size Fits All: An Idea Whose Time has Come and Gone”. In: Proceedings of the 21st International Conference on Data Engineering (ICDE), 2005, pp. 2-11.
- [St07] Stonebraker, M., et al: The End of an Architectural Era (It's Time for a Complete Rewrite), Proceedings of VLDB '07, 2007, Wien, pp. 1150-1160,.

Demonstrationsprogramm

Improving Service Discovery through Enriched Service Descriptions

Mohammed AbuJarour, Felix Naumann
Hasso-Plattner-Institut
University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

Abstract:

The increasing popularity of the Software-as-a-Service and Cloud Computing trends has been among the main factors behind the *increasing number of public web services* in several domains, e.g., e-commerce, enterprise, education, government, etc. Moreover, the functionalities of such web services are becoming *more complex* due to the complexities of modern business needs and marketplaces. Additionally, it has been observed that service providers, who represent the single source of information about web services, typically release *poor service descriptions*.

Due to the aforementioned factors, service discovery has become one of the main challenges in Service-oriented Computing (SOC). In this demo, we show how to enrich service descriptions enabling enhanced service discovery. In our approach, web services are enriched with annotations (textual descriptions and tags) that are automatically extracted from the websites of their providers and from the analysis of their invocations.

1 Web Service Discovery

The problem of web service discovery is similar to looking for a needle in a haystack. Seeking the right service based on user's search criteria is still one of the main challenges in Service-oriented Computing (SOC). Typically, service descriptions released by service providers are used to perform service discovery. However, several factors exacerbate this challenge, such as the increasing number of public web services, their complex functionalities, and poor service descriptions. Further details on this problem and its related literature are provided in [AN10a].

In spite of their crucial role in SOC, researchers have identified several limitations in service descriptions. For instance, the "Internet of Services" (IoS) vision introduced the concept of *business services* [SAP09]. Business services represent an abstraction of the IT web services. Their requirements need much information about the considered services rather than the technical information provided by service providers in the form of service descriptions. In [SAP09], the authors state that "... there is definitely the need for more than the technical description of a web service interface".

To handle the challenge of poor service descriptions that are not suitable for service discovery, we use additional sources of information to enrich them. These sources are automatic

annotations based on the providers' websites and invocation analysis.

This demo is part of our Depot project [ACM⁺09, AN10b].

2 Enriching Service Descriptions

Enriched service descriptions have many benefits in SOC, e.g., enhanced service discovery. Several approaches have been proposed to enrich service descriptions. For instance, the Adaptive Service Grid (ASG) project¹ enables domain experts to enrich service descriptions with ontology annotations *manually*. In our approach, we enrich service descriptions automatically with two types of additional information, namely, service annotations and invocation analysis.

2.1 Annotating Web Services

Along with the technical service descriptions (published in service registries) that service providers release about their web services, they give additional textual descriptions (usually on their own websites) to explain their functionalities. Typically, such textual descriptions do not appear in their counterpart technical service descriptions. We developed a *focused crawler* to collect public web services from the Internet automatically [ACM⁺09]. Using only the collected technical service descriptions to perform service discovery was not efficient, because they are typically poor. To enhance service discovery, we introduced an *information parser* to extract additional information about the collected web services from the crawled websites [ANC10].

Two types of information are generated using our information parser, namely, textual annotations and tags. Text in a webpage that is close to a reference to a web service is extracted as an annotation to that web service. The entire content of a webpage where a web service is referenced is used to generate tags for that web service. These generated tags are then used to help service consumers browse web services through tag clouds.

2.2 Dynamic Tags via Invocation Analysis

Invocation analysis is an additional source of information about web services. This source is instance-based, where actual service invocations are used to generate tags for the invoked web services. In our approach, we consider dynamic data web services only, such as news, events, promotions and offers, etc. For such web services, dynamic tags are generated based on the analysis of their invocations [AN10a]. These tags are integrated with the tags generated by the information parser to provide a unified tag cloud.

¹ASG Project: <http://www.asg-platform.org>

3 Enhanced Service Discovery

Based on the enriched service descriptions, four types of service discovery are provided on our platform:

1. Browse by category: The increasing complexity of web services and their driving business needs makes finding “good” keywords for full-text search a difficult task. For such cases, we provide web service browsing based on categories. Collected web services are automatically classified in several application domains, e.g., education, finance, entertainment, etc. This classification is based on the enriched descriptions of web services.
2. Browse by tag cloud: For a quick way of exploring common web services, regardless of their providers or categories, we provide a tag cloud that enables service consumers to browse through common tags attached to web services. Part of these tags are automatically generated from websites of service providers during service crawling through the information parser. Additional dynamic tags are generated from invocation analysis of dynamic data web services.
3. Full-text search: This type requires basic knowledge in the application domain to choose “good” keywords, e.g., address normalization, credit card validation, etc. Figure 1 shows a screenshot of our search interface.
4. Browse by provider: This type of service exploration enables service consumers to find relevant web services from specific service providers. For instance, service consumers prefer to use web services from service providers with high reputation or well-known providers.

4 The Demonstration

Depot allows service providers to register their web services explicitly. Additionally, we allow service consumers to suggest web services by providing the URL of their provider. Depot crawls that URL, collects web services provided on that URL, extracts annotations for the collected web services from the same URL, and classifies them based on the extracted annotations. For a small website with a few HTML pages and a few web services, these steps take a couple of minutes.

In this demo, we show how Depot collects public web services released on a provided URL and annotates them. Based on these annotations, Depot derives classifications for these web services automatically. We show how service consumers can then browse web services based on their provider, category, tags, or annotations (keyword-based).

The screenshot displays the Posr web interface. At the top, the logo 'Posr' is followed by navigation links: [trans.posr](#), [pro.posr](#), [ex.posr](#), [sup.posr](#), and [com.posr](#). Below these are links for [Crawl](#), [Crawled URL's](#), [Find](#), [Browse](#), [Service Tags](#), [New Provider](#), and [New Service](#). On the left, there are links for [HPI](#), [Contact](#), and [Imprint](#). The main content area shows search results for the keyword 'gene'. It lists three services found: (1) **Ensembl**, (2) **NCBIGenomeAnnotation**, and (3) **GetEntry**. Each service entry includes a 'View WSDL' link, a list of operations (e.g., 'get version', 'get gene list'), and a 'View Operations' button. The services are provided by the 'HPI Team' and have a category of 'None'. On the far left, there is a sidebar with 'Selected Operations' and a list of services including 'FileTransferService', 'ABRXMLSearchRPC', 'ZipCity', and 'GlobalAddressInteractiveVerific'.

Figure 1: A screenshot of our full-text search interface. Available online at: <https://www.hpi.uni-potsdam.de/naumann/sites/servicedepot/>

References

- [ACM⁺09] Mohammed AbuJarour, Mircea Craculeac, Falko Menge, Tobias Vogel, and Jan-Felix Schwarz. Posr: A Comprehensive System for Aggregating and Using Web Services. In *SERVICES '09: Proceedings of the 2009 Congress on Services - I*, pages 139–146, Los Angeles, CA, USA, 2009. IEEE Computer Society.
- [AN10a] Mohammed AbuJarour and Felix Naumann. Dynamic Tags For Dynamic Data Web Services. In *Workshop on Enhanced Web Service Technologies*, Ayia Napa, Cyprus, 2010. ACM.
- [AN10b] Mohammed AbuJarour and Felix Naumann. Information integration in Service-oriented Computing. In *Ph.D. Symposium at the European Conference on Web Services*, Ayia Napa, Cyprus, 2010.
- [ANC10] Mohammed AbuJarour, Felix Naumann, and Mircea Craculeac. Collecting, Annotating, and Classifying Public Web Services. In *OTM 2010 Conferences*, Crete, Greece, 2010. Springer.
- [SAP09] SAP Research. Unified Service Description Language. <http://www.internet-of-services.com/uploads/media/USDL-Information-Sheet.pdf>, 2009.

StreamCars – Datenstrommanagementbasierte Verarbeitung von Sensordaten im Fahrzeug

André Bolles, Dennis Geesen,
Marco Grawunder, Jonas Jacobi,
Daniela Nicklas, H.-Jürgen Appelrath
Universität Oldenburg
vorname.nachname@uni-oldenburg.de

Marco Hannibal, Frank Köster
Deutsches Zentrum für
Luft- und Raumfahrt
Braunschweig
vorname.nachname@dlr.de

Abstract: Wir präsentieren StreamCars, eine datenstrommanagementbasierte Architektur für die flexible Verarbeitung von Sensordaten im Fahrzeug. Das Datenstrommanagementsystem Odysseus stellt Anfrageverarbeitungsmechanismen zur Objektverfolgung und zum Aufbau eines Kontextmodells bereit. Auf dieses wird dann über kontinuierliche Datenstromanfragen zugegriffen, sodass spezifische Informationen an unterschiedliche Assistenzfunktionen in Fahrzeugen weitergeleitet werden können.

1 Motivation

Moderne Fahrerassistenzsysteme (FAS) wie bpsw. ein adaptiver Tempomant berücksichtigen neben dem eigenen Fahrzeugzustand auch Objekte wie vorausfahrende Fahrzeuge. Solche Fahrerassistenzsysteme werden in der Regel als proprietäre Systeme entwickelt, die hochspeziell auf Eigenschaften der Assistenzfunktion oder auch der eingesetzten Sensorik (Radar, Video, etc.) abgestimmt sind. So können zwar die Sensordaten effizient verarbeitet werden, aber die Wartbarkeit und Erweiterbarkeit der Systeme ist eingeschränkt. Anpassungen an neue Anforderungen lassen sich in diesen Systemen oft nur durch eine manuelle Reimplementierung großer Teile der Software realisieren. Da Änderungen insbesondere während der Entwicklung neuartiger Assistenzsysteme sehr oft vorkommen, ist hier eine flexiblere Architektur für solche Systeme notwendig.

Die Idee dieser Arbeit ist es, ein Datenstrommanagementsystem (DSMS) mit seinen Anfrageverarbeitungsmechanismen als Basis einer anpassbaren und erweiterbaren Architektur für FAS einzusetzen und so die Entwicklungs- und Integrationskosten von FAS zu senken. Dabei übernimmt das DSMS eine Art Middleware-Rolle zwischen der Sensorschicht und der eigentlichen Assistenzfunktion. Zum einen werden Datenstromanfragen zur Aufbereitung der vorverarbeiteten Sensordaten zu einem Kontextmodell eingesetzt und zum anderen bietet das DSMS den einzelnen Assistenzfunktionen einen anfragegesteuerten Zugriff auf diese Daten.

Diese Arbeit bringt die zwei Forschungsfelder FAS und Datenstrommanagement (DSM) zusammen. Im Bereich der FAS existieren bereits zahlreiche Arbeiten, die aber vornehmlich die intelligente Nutzung spezieller Sensorik fokussieren [GF07, ICPRK08]. Die Betrachtung von flexiblen Architekturkonzepten, welche sowohl von der Sensorik als auch

von der Assistenzfunktion abstrahieren, wird dagegen nur vereinzelt herangezogen [G⁺08, DW05]. So wird in [GF07] bspw. ein System entwickelt, das speziell auf verschiedene Radarsensoren am Fahrzeug ausgerichtet ist. Eine Architekturbetrachtung der entwickelten Software erfolgt nur am Rande. [G⁺08] beschreibt dagegen DOMINION, eine serviceorientierte Plattform, die von der eingesetzten Sensorik und der Fahrzeughardware abstrahiert jedoch nicht die eigentliche Sensordatenverarbeitung beinhaltet.

Im Bereich des DSM existieren zahlreiche Arbeiten (u. a. [KS09]), in denen prototypische DSMS in speziellen Anwendungsfeldern wie Finanztransaktionen oder Click-Streams entwickelt werden. In diesem Demonstrator füllt Odysseus [BGJ⁺09] die Lücke zwischen der Sensordatenvorverarbeitung und der Assistenzfunktion im Fahrzeug.

2 Architektur

Das System ist als Schichten-Architektur implementiert. Die Aufteilung der Schichten ist dabei an [DW05] angelehnt und definiert über der Sensorik eine Sensordatenschicht, eine Objektdatenschicht und eine Anwendungsschicht, die dann zurück zur Fahrzeugelektronik führt (s. Abbildung 1). Anders als in [DW05] ist hier die Objektdatenschicht jedoch nicht als proprietäres System implementiert, sondern wird durch das DSMS Odysseus gefüllt. So wird eine flexible Aufbereitung der Daten für verschiedene Assistenzfunktionen unter Einsatz verschiedener Sensoren möglich.

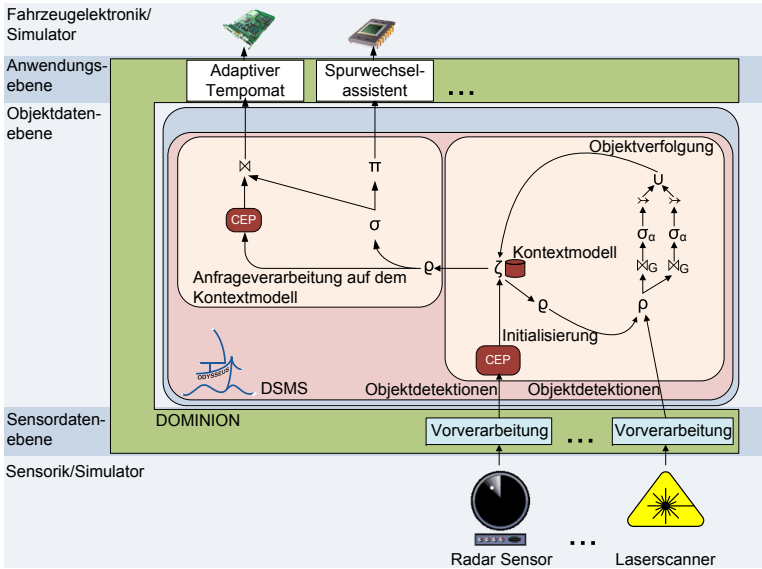


Abbildung 1: StreamCars-Architektur

Die **Sensorik** oder im simulierten Testfall der Simulator erfasst das Umfeld entsprechend ihrer physikalischen Möglichkeiten.

Auf der **Sensordatenebene** werden innerhalb von DOMINION die Sensorrohdaten vorverarbeitet. Hier können bspw. Verfahren zur Bildverarbeitung oder Punktwolkensegmentierung eingesetzt werden, um aus den Sensorrohdaten Featurevektoren mit Informationen wie Geschwindigkeit, Position oder der Bounding Box der detektierten Objekte zu extrahieren. Diese Featurevektoren werden anschließend an das DSMS Odysseus weitergeleitet.

Odysseus übernimmt auf der **Objektdatenebene** in dieser Architektur zwei zentrale Aufgaben. Zum einen setzt es eine *Objektverfolgung* durch einen geeigneten Anfrageplan um. Hierzu wurden für Odysseus entsprechende Operatoren zur Prädiktion, Assoziation und Filterung von Objekten entwickelt und integriert. Diese Operatoren sind so gestaltet, dass auch hochspezielle Verfahren, die auf bestimmte Sensoren abgestimmt sind, implementiert und in einen Anfrageplan eingebaut werden können. Auch garantiert die definierte Semantik der Operatoren deterministische Ergebnisse der Anfrage. Die detektierten Objekte werden in einem Kontextmodell gespeichert, das laufend aktualisiert wird. *Anfragen auf dem Kontextmodell* liefern im zweiten Schritt die Daten, die eine Assistenzfunktion zur Ausführung ihrer Aufgabe benötigt. Hier können die Daten beliebig für Assistenzfunktionen aufbereitet werden. Durch die Integration aller Sensorinformationen in ein gemeinsames Kontextmodell können, anders als in den meisten proprietären Systemen, auch integrierte Assistenzfunktionen entwickelt werden. Beispielsweise sollte ein adaptiver Tempomat das Fahrzeug nicht abbremsen, wenn der Spurwechselassistent erkennt, dass ein Spurwechsel erfolgen soll.

Auf der **Anwendungsebene** werden in DOMINION die Assistenzfunktionen selbst implementiert. Hier ist eine Ansteuerung der Fahrzeugelektronik möglich.

Die **Fahrzeugelektronik** bzw. im simulierten Testfall der Simulator setzt die Befehle der Assistenzfunktion um und greift steuernd in das Fahrzeugverhalten ein.

3 Demonstration

Der Demonstrator ist an eine Fahrsimulatorsoftware angeschlossen, die die benötigten Sensordaten liefert. Es ist möglich über die Benutzeroberfläche von Odysseus Anfragen an die Simulationssoftware zu stellen. Diese Anfragen dienen zum einen der Aufbereitung des Kontextmodells und können zum anderen dazu genutzt werden, das Verhalten des Fahrzeugs bzw. der Assistenzfunktion zu beeinflussen. Aktuell ist ein adaptiver Tempomat implementiert. Der Besucher des Demonstrationsstandes wird die Möglichkeit haben, das Fahrzeug im Fahrsimulator zu steuern und zu sehen, wie sich das Verhalten des Fahrzeugs ändert, wenn die Anfragen an das System verändert werden. Die Abbildung 3 zeigt die Benutzeroberfläche von Odysseus sowie unten rechts den Radarkegel des simulierten Radarsensors in der Simulationssoftware.

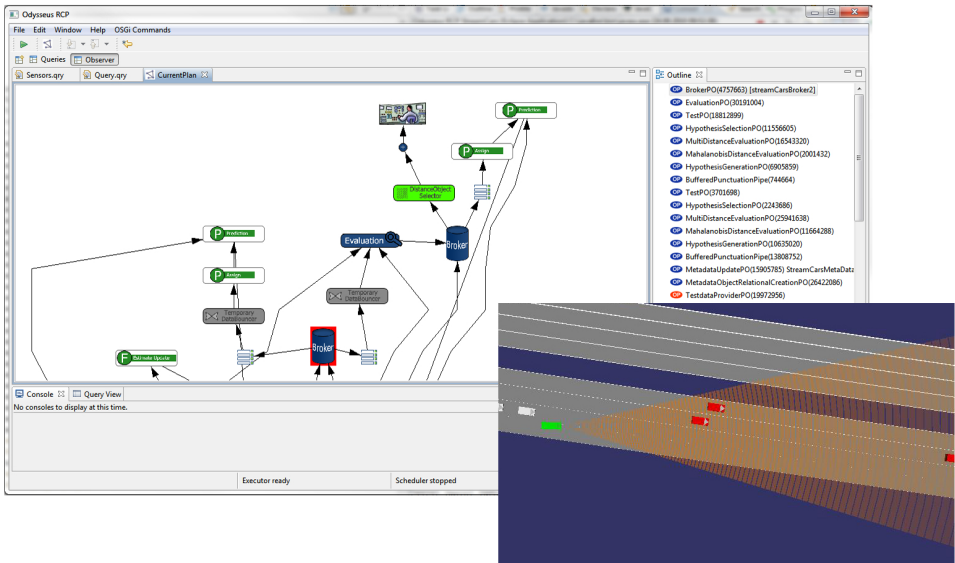


Abbildung 2: Screenshots von Odysseus und der Simulationssoftware

Literatur

- [BGJ⁺09] Andre Bolles, Marco Grawunder, Jonas Jacobi, Daniela Nickla und H.-Jürgen Apperath. Odysseus: Ein Framework für maßgeschneiderte Datenstrommanagementsysteme. In *Informatik 2009*, 2009.
- [DW05] Michael Darms und Hermann Winner. A modular system architecture for sensor data processing of ADAS applications. In *Proceedings of the IEEE Intelligent Vehicles Symposium 2005*, 2005.
- [G⁺08] Jan Gacnik et al. Service-Oriented Architecture for Future Driver Assistance Systems. In *FISITA 2008*, 2008.
- [GF07] Mahdi Rezaei Ghahroudi und Alireza Fasih. A Hybrid Method in Driver and Multisensor Data Fusion, Using a Fuzzy Logic Supervisor for Vehicle Intelligence. In *Proc. of the Int. Conf. on Sensor Technologies and Applications 2007*. IEEE Computer Society, 2007.
- [KS09] Jürgen Krämer und Bernhard Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34(1), 2009.
- [ICPRK08] Álvaro Catalá-Prat, Ralf Reulke und Frank Köster. Early detection of hazards in driving situations through multi-sensor fusion. In *FISITA 2008*, 2008.

NexusDSEditor — Integrated Tool Support for the Data Stream Processing Middleware NexusDS

Nazario Cipriani, Carlos Lübke and Oliver Dörler
IPVS - Universität Stuttgart

{nazario.cipriani | carlos.luebke}@ipvs.uni-stuttgart.de, oliver@doerler.name

Abstract: In this paper we present NexusDSEditor — an integrated tool for the stream processing middleware NexusDS. NexusDSEditor is an extension module for the NexusEditor and supports developers with designing new streaming applications by providing an integrated tool for orchestrating stream query graphs, define the deployment of query graph fragments to execution nodes, and analyzing data streams. In this paper we demonstrate these single steps and show how NexusDSEditor supports developing streaming data applications for the NexusDS platform by hiding complexity and providing an intuitive user interface.

1 Motivation

Context-based applications access data regarding the user's current situation to adapt accordingly [Dey01]. As an example consider the visualization pipeline as described in [CLM10]. A continuous stream of data flows through the visualization pipeline and results in a rendered image of a map showing nearby surroundings received by a client application. The application helps the user by adapting its behavior to the user context. To process such highly specific schemes, NexusDS [CEB⁺09] has been developed. NexusDS is a highly flexible and extensible stream processing middleware targeting the processing of context data streams in a highly distributed and heterogeneous environment. Nexus [NGS⁺01] as well as NexusDS build up a context data management platform where Nexus offers access to static context data and NexusDS provides access to streamed context data. The context data management platform is an open, federated platform for mobile, context-aware applications where arbitrary data providers can make their data available by the platform. A central component is the Augmented World Model (or short AWM) [NM04]. The AWM is a shared, global context model which can be extended by extension schemas.

In the past decade many data stream processing systems have been proposed. However, tool support for those systems is not considered by research community. An integrated tool that supports developers of streaming applications, when designing new or modifying existing ones, is beneficial. It helps reducing development time and prevent errors that occur at design time. As presented in [CWGN10], an integrated tool which hides the real task complexity and supports the developer during design time is highly beneficial. As a consequence, we have integrated support for the development of streaming applications within the NexusEditor [NN08]. Thereby, several requirements specific to the domain of

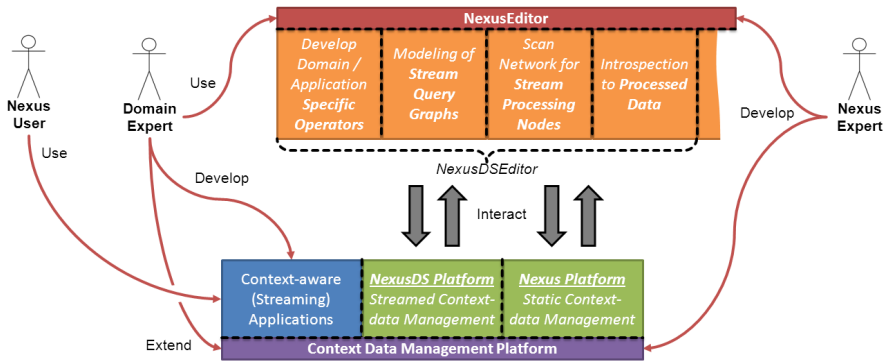


Figure 1: Embedding of the NexusDSEditor

streaming applications must be met: Support *development of new domain and application specific operators*, provide an intuitive graphical user interface that supports *definition of stream queries*, offer the possibility to *scan networks for available stream processing nodes* operators can be deployed to, and present a way for getting an *introspection of data processed by the stream query graph* at runtime.

2 Architecture

Figure 1 shows the embedding of the NexusDSEditor within the Nexus system. The NexusEditor is the central component in supporting the development process and bridging the world of *Nexus Experts* and *Domain Experts*.

On the right, Nexus Experts develop and maintain the context data management platform as well as the NexusEditor. The context management platform consists of the Nexus [NGS⁺01] and the NexusDS [CEB⁺09] platforms. Nexus Experts also develop extensions for the NexusEditor, such as the NexusDSEditor extension as displayed in Figure 1. NexusDSEditor supports Domain Experts during the development process of context-aware streaming applications (as we will show in our demonstration).

On the left, Domain Experts exploit the NexusEditor functionality, and here especially the NexusDSEditor functionality, to develop context-aware applications and context data management platform extensions respectively. By using the NexusDSEditor, productivity is increased since most potential conflicts are recognized and can be eliminated at design time. E.g., beside others the NexusDSEditor supports compatibility checking of interconnections between operators to guarantee a working query graph at design time.

Finally Users use context-aware applications and access functionality and data which Domain Experts have developed beforehand. These context-aware applications run on clients, such as desktop computers or mobile devices.

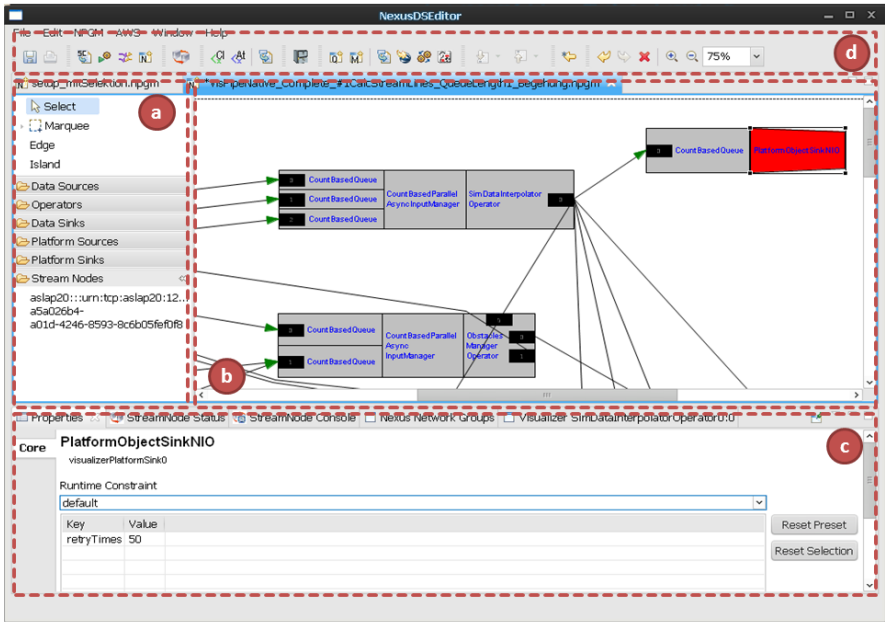


Figure 2: NexusDSEditor screenshot

3 Demonstration

Figure 2 shows a screenshot of the NexusDSEditor . On area (a) resides the toolbox offering different operators and (b) is the drawing area where the orchestration of the actual stream query graph is performed. (c) represents a properties view dependent on the currently selected item from (b). Finally, (d) offers some shortcuts to mostly used functionality grouped within a toolbar.

We demonstrate four scenarios, each focusing on a different requirement from Section 1:

Development of new domain and application specific operators Developers have the possibility to integrate specific operators in NexusDS. To do so, the developer has to provide the actual implementation of the physical operator and also a set of meta data that describes the operator’s properties. To satisfy this requirement, NexusDS-Editor provides a modeling component [CWGN10] that allows to easily model the required operator meta data and package the operator for later deployment.

Definition of stream queries NexusDSEditor provides a graphical interface to orchestrate the stream query graph out of single units called operators. An operator is the basic concept of stream query graphs and represents a certain operation on the streamed data. After the stream query graph is modeled, NexusDSEditor supports

the fragmentation of the stream query graphs to so-called *isles*. An isle is a fragment of a stream query graph that is deployed on a stream processing node.

Scan network for available stream processing nodes In NexusDS, stream query graphs are processed in a distributed fashion. Each node, that runs the NexusDS system, can join and consequently be used as stream processing node running a stream query fragment. NexusDSEditor supports scanning the network for available stream processing nodes and present them to the developer. In a second stage the developer can pick the stream processing nodes he prefers and assign them to the corresponding isles. After that process, the query graph is fully defined and ready for deployment. NexusDSEditor therefore offers a deployment functionality that deploys the isles to their associated stream processing nodes initiating the query execution.

Introspection of data processed by the stream query graph After the query graph has been deployed, it is executed and data is processed. Nevertheless, errors may occur during operator development. Therefore it is beneficial, if developers can analyse processed data that is transferred between operators. For this purpose, NexusDS-Editor supports a special kind of operator class namely *Visualizers*. Visualizers consist of two components: One component for connecting to the query graph to retrieve the corresponding data and a second component to display the data within the NexusDSEditor.

References

- [CEB⁺09] Nazario Cipriani, Mike Eissele, Andreas Brodt, Matthias Grossmann, and Bernhard Mitschang. NexusDS: A Flexible and Extensible Middleware for Distributed Stream Processing. In *IDEAS '09: Proceedings of the 2008 international symposium on Database engineering & applications*, pages 1–8, New York, NY, USA, 2009. ACM.
- [CLM10] Nazario Cipriani, Carlos Lübke, and Alexander Moosbrugger. Exploiting Constraints to Build a Flexible and Extensible Data Stream Processing Middleware. In *The Third International Workshop on Scalable Stream Processing Systems*, Atlanta, USA, 4 2010.
- [CWGN10] N. Cipriani, M. Wieland, M. Großmann, and D. Nicklas. Tool support for the design and management of context models. *Information Systems*, In Press, Corrected Proof:–, 2010.
- [Dey01] Anind K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.
- [NGS⁺01] Daniela Nicklas, Matthias Großmann, Thomas Schwarz, Steffen Volz, and Bernhard Mitschang. A Model-Based, Open Architecture for Mobile, Spatially Aware Applications. In *SSTD*, pages 117–135, 2001.
- [NM04] D. Nicklas and B. Mitschang. On building location aware applications using an open platform based on the NEXUS Augmented World Model. *Software and System Modeling*, 3:303–313, 2004.
- [NN08] Daniela Nicklas and Carsten Neumann. NexusEditor: A Schema-Aware Graphical User Interface for Managing Spatial Context Models. In *MDM*, pages 213–214, 2008.

AIMS: An SQL-based System for Airspace Monitoring

Gereon Schüller, Andreas Behrend

{gereon.schueller@fkie.fraunhofer.de, behrend@cs.uni-bonn.de}

Fraunhofer FKIE, Dept. Sensor Data Fusion

Neuenahrer Straße 20, D-53343 Wachtberg, Germany

Abstract: The “Airspace Monitoring System” (AIMS) is a system for monitoring and analyzing flight data streams with respect to the occurrence of arbitrary complex events. It is a general system that allows for a comprehensive analysis of aircraft movements, in contrast to already existing tools which focus on a single task like flight delay detection. For instance, the system is able to detect critical deviations from the current flight plan, abnormal approach parameters of landing flights as well as areas with an increased risk of collisions. To this end, tracks are extracted from cluttered radar data and SQL views are employed for a timely processing of these tracks. Additionally, the data is stored for later analysis.

1 Tracking Flights

Airspace monitoring systems visualize the current state of aircrafts based on radar observation or on transponder signals, or both. However, there are still plenty of situations where highly critical events are detected too late, like the Ueberlingen catastrophe has shown [4] that occurred due to the inattentiveness of the responsible flight controller. There are many critical events in airspace, like close encounters or violations of no-fly zones. Disappearance of flights from their scheduled route are another anomaly of interest; they can indicate hijacking of airplanes. The development of automated tools able to detect critical events based on streams of continuously arriving sensor data and to raise an alarm triggering human (or automatic) action is therefore a highly relevant, but still mostly unsolved task. First of all, it is necessary to track the airplanes. A *flight track* is a sequence of probability density functions representing an estimation of a flight path. It is derived from measurements either by primary radar or by transponder signals of equipped aircrafts. But the raw position measurements suffer from several drawbacks: Radar or transponder signals may be missed due to physical effects, and false detections may arise from punctual disturbing sources like clouds, birds etc. Misdetetection and false alerts/clutter are connected anti-proportionally as leveraging the amplification of the signal will also amplify the disturbing signals [5]. Another problem consists in the fact that radar signals are not sharp in position and will be smeared out. If multiple objects are in the field of view, the assignment between plot and corresponding plane has to be solved.

All these effects make the process of tracking complex. Many tracking algorithms have been proposed in the past decade, e.g. [6, 7, 8]. The development of new and enhanced tracking algorithms is still an active research area, in particular at Fraunhofer FKIE. Track-

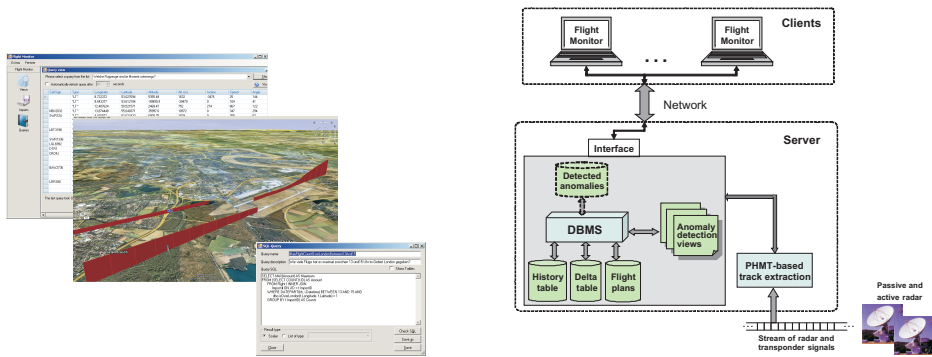


Figure 1: GUI of the Flight Monitor (left), Architecture of AIMS (right)

ing algorithms can be distinguished by several characteristics. For example, there are trackers that can cope with multiple targets and will compute several hypotheses about which plot belongs to which target. Other trackers feature prediction, estimating how the target will behave in future, or they may feature retrodiction, calculating what we can say about the way in the past by knowing the current results. In our approach, a Bayesian tracking algorithm is employed. It is important to know that trackers usually use stochastic models and will output the kinematic state together with a quality indicator of the estimation, i.e., the covariance in position and speed as well as a measure for the size of the region in which the object can be estimated with a certain probability. Thus, a sample output of our tracking algorithm may look as follows:

timestamp	id	lon	lat	height	covX	covY	vel	heading
19:20:43	34	51.124	7.054	4534	30.2	1.2	300.4	91
19:20:44	35	50.985	6.345	2324	40.4	1.4	240.3	27
...

2 The AIMS System

AIMS is a prototype of a system for monitoring and analyzing flight movements. It has been developed at the University of Bonn in cooperation with Fraunhofer FKIE and EADS Deutschland GmbH. The aim is to develop efficient DBMS-based methods for real-time gathering and monitoring of streams of flight data. Currently, the system is used to monitor the complete German airspace every 4 seconds with up to 2000 flights in peak times. An overview of the AIMS architecture is given in Figure 1.

Our system basically consists of three components. The first one is a *track extraction* tool used for detecting and identifying aircrafts using radar data and transponder signals. Since radar data usually contain noise, we employ the *probabilistic multiple hypothesis*

tracking (PMHT) algorithm for detecting moving objects in a cluttered environment [8]. The PMHT algorithm calculates a sequential likelihood ratio for solving the plot-to-track association problem, testing the hypothesis that a plot belongs to a track. The resulting tracks and the transponder signals are then merged into a stream of highly accurate aircraft positions. The resulting stream of timestamped position and velocity data (track data for short in the following) is periodically pushed into a relational database. For programming this database application, we used Microsoft SQL Server because of its well-optimized user-defined function (UDF) processing. UDFs are, e.g., used for spatial coordinate transformations and vector computations. In our scenario, new track data is provided every four seconds and stored in a 'delta table' containing just the most recent track data. Its former content is moved to the history table such that the complete course of each flight is recorded. However, before data is stored in the database, a lot of data transformation and cleansing has to be done for which we employ SQL triggers.

The main feature of our approach to stream monitoring is to use continuous SQL queries (stored as views) for specifying anomalous situations to be detected, as well as important application-specific concepts and parameters required for anomaly detection. A similar approach has been already used in various other successful applications by our group ([1, 2]). In the flight tracking scenario, the following questions are to be answered by the AIMS system:

1. Which aircrafts are currently airborne?
2. Which aircrafts are currently landing?
3. Are there aircrafts on collision course?
4. Are there critical deviations from flight plans?
5. How many aircrafts are currently over a certain region?
6. What is the average number of landings for an arbitrarily chosen airport?

The view definitions can be created in the so-called *Flight Monitor*, a graphical user interface written in Visual Basic and C# that allows the user to interact freely with the system (see Fig. 1). Its main tasks are the support for defining new anomaly detection views (although the resulting views are then stored in and managed by the relational database system) and the continuous textual and graphical monitoring of their results. The incorporated SQL editor allows to freely define new detection views which may directly access the underlying tables and/or other anomaly detection views already defined. In this way, various conditions of given detection views can be combined in order to define complex events with respect to critical aircraft movements.

In addition, the Flight Monitor conducts performance measurements for each periodic query execution. After considerable tuning, all of the above queries could be executed in less than 2 seconds (except for the last one determining average values), being below the refreshment rate of 4 seconds. Another feature of the Flight Monitor is its graphical visualization of the detected anomalies by means of an exported KML (keyhole markup language) file which can be processed by several programs. In our case, Google Earth is employed as long as our system remains in a prototypical status. In order to achieve a meaningful graphical visualization, the anomaly detection views usually retain attributes for position and time values of the flight data under consideration.

3 Incremental Materialization for Performance Enhancement

Even though a considerable degree of analysis can be achieved by purely recomputing expressive SQL views in each refreshment cycle, the given stream scenario will sooner or later drastically slow down our system without further optimizations.

To this end, we use an incremental approach to materialize the running queries. The track data are separated into two relations; one for the data of the current sliding window, the other for older track data. While most queries will only run on the newest data, some queries will include the usage of historic track data, but only those who are related to the newest data, allowing for the usage of joins between new and old data. Index structures are used to support fast access to these historical data. The update propagation process, as well as moving data into the relation for historical data is done by a system of cascading triggers that fire on the input of arriving data. This way, the views are always up to date after insertions. Some queries do not fall into these categories, using large portions of historical track data, e. g. "What is the average number of landings for an arbitrarily chosen airport?". However, the nature of these queries is not a running one, they are merely use for statistics and are executed infrequently.

First evaluation results already indicate that conventional SQL queries can be used for efficiently processing this realistic stream scenario [3]. Since AIMS is still in a prototypical state, however, a comprehensive performance evaluation cannot be presented yet. However, all presented anomaly detection views - which include the determination of landing and departing flights, critical approaches, deviations from flight plans as well as the determination of delay times - could be executed in less than 2 seconds. AIMS provides a more flexible approach to airspace monitoring allowing the free definition of arbitrary complex events over a stream of flight data. The flexibility results from using SQL views which freely add and combine user-defined anomaly detection view specifications.

References

- [1] A. BEHREND, C. DORAU, R. MANTHEY, AND G. SCHÜLLER: *Incremental view-based Analysis of Stock Market Data Streams*. IDEAS 2008: 269-275.
- [2] A. BEHREND, R. MANTHEY, G. SCHÜLLER, AND MONIKA WIENEKE: *Detecting Moving Objects in Noisy Radar Data Using a Relational Database*. ADBIS 2009: 286-300
- [3] G. SCHÜLLER, A. BEHREND, AND R. MANTHEY: *AIMS: An SQL-Based System for Airspace Monitoring*. IWGS 2010: 31-38
- [4] BUNDESSTELLE FÜR FLUGUNFALLUNTERSUCHUNG: *Investigation Report AX001-1-2/02*, May 2004.
- [5] M. I. SKOLNIK: *Introduction to Radar Systems*. McGraw-Hill, 1981.
- [6] S. S. BLACKMANN AND R. POPOLI: *Design and Analysis of Modern Tracking Systems*. Artech House, Boston, USA, 1999.
- [7] G. VAN KEUK. Mht Extraction and Track Maintenance of a Target Formation. *Aerospace and Electronic Systems, IEEE Transactions on*, 38(1):288–295, Jan 2002.
- [8] M. WIENEKE, W. KOCH: On Sequential Track Extraction within the PMHT Framework. *EURASIP Journal on Advances in Signal Processing*, 2008: 1–13.

PROOF¹: Produktmonitoring im Web

Christian Wartner, Sven Kitschke

WDI-Lab, Institut für Informatik, Universität Leipzig
{wartner | kitschke}@informatik.uni-leipzig.de

Abstract: Für Verbraucher, Händler und Hersteller ist die Beobachtung der Entwicklung von Angebotspreisen interessant, seien dies Preise von Produkten, Flügen oder Dienstleistungen. Wir präsentieren mit PROOF einen neuartigen Ansatz Produktmonitoring durchzuführen. PROOF ist ein erweiterbares System zum Integrieren und Analysieren von Webdaten. Die definierbaren Workflows erlauben unter anderem Anfrageoptimierungs- und Objekt-Matching-Operationen, um eine hohe Datenqualität bei guter Performance zu erreichen.

1 Motivation

Der Onlinehandel mit Produkten ist in den vergangenen Jahren stetig gewachsen. Neben Online-Shops sind über 1000 Preisvergleichsportale entstanden [KH07], welche die Angebote der Händler aggregieren. Um einen möglichst vollständigen Überblick über Produktangebote zu erhalten, ist man gezwungen, viele Shops zu besuchen oder sich mehrerer Preisvergleichsportale zu bedienen. Wenn Angebote eines Händlers in unterschiedlichen Portalen gefunden werden, sind die gewonnenen Ergebnisse um Duplikate zu bereinigen. Will man die Preisentwicklung über einen längeren Zeitraum beobachten, muss diese Aufgabe zudem häufig wiederholt werden.

Wir demonstrieren mit unserer Applikation, dass das Auffinden und die Identifikation relevanter Datensätze zu Produktangeboten, das Zusammenführen dieser oft heterogenen Datensätze und ihre Analyse in vielen Teilen automatisiert und damit effizienter gestaltet werden kann. Produktmonitoring mit PROOF besitzt folgende Vorteile:

- Skriptbasierte Steuerung der Workflows für Erweiterbarkeit, Adaption an konkrete Analyseaufgaben und Einsatz in mehreren Domänen (z.B. auch für Flüge und Immobilien),
- Querygeneratoren [ETR09] zur Anpassung an unterschiedliche Anfrageschnittstellen und zur Optimierung hinsichtlich Qualität und Effizienz,
- flexible Objekt-Matching-Verfahren zur Duplikatbereinigung,
- Operatoren zur effizienten Verarbeitung heterogener Daten.

¹ Product Offer Fusor

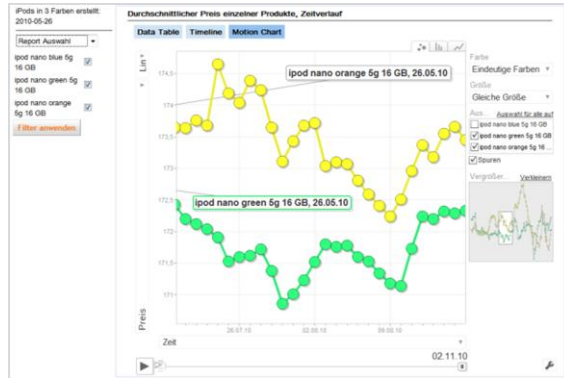


Abbildung 1: Screenshot – Darstellung der Preisentwicklung zweier Produkte

Die periodisch ausgeführte Aggregation der Angebote befreit den Nutzer von der wiederkehrenden und zeitraubenden Aufgabe, verschiedene Webshops und Preisvergleichsportale zu besuchen und deren Angebote und Produktpreise manuell zu extrahieren. Langfristig von Vorteil ist, dass der Workflow für andere Produkte wiederverwendet werden kann und lediglich parametrisiert werden muss.

2 Unser Ansatz

Architektur: Bei der vorgestellten Anwendung handelt es sich um eine AJAX²-basierte Webapplikation. Der im Webbrowser ausgeführte clientseitige Teil der Anwendung dient zum Erstellen und Verwalten von *Monitoring-Jobs* sowie zur Visualisierung und Auswertung der gesammelten Daten. Aufgabe des Servers ist die periodische Ausführung von *Extraktionsworkflows* und das Bereitstellen von aus den gesammelten Daten erstellten Reports. Die dazu nötigen Operationen werden mit Hilfe von *weFuice*, einer Weiterentwicklung von *iFuice*³ [Ra05], realisiert. *weFuice* stellt eine Skriptsprache zum Definieren von Datenintegrationsworkflows bereit. In ihr werden Operatoren für die Generierung von Anfragen, zum Zugriff auf verschiedenste Datenquellen, zum Objekt-Matching sowie für weitere Funktionen wie z.B. Mengenoperationen und statistische Funktionen genutzt.

Ansatz: Ausgangspunkt für das Produktmonitoring ist das Erstellen von *Monitoring-Jobs*, dem eine vom Benutzer zu erstellende Liste von Produkten zu Grunde liegt. Für jeden *Monitoring-Job* werden ein Extraktionsintervall und ein *Extraktionsskript* festgelegt. Das Sammeln von Daten zu Produktangeboten erfolgt durch die periodische Ausführung des *Extraktionsskriptes* (siehe Abb. 2 links), welches Anfragen aus der Liste der zu überwachenden Objekte an *Monitoring-Datenquellen*, wie Preisvergleichsseiten oder Metasuchmaschinen, generiert, ausführt und deren Ergebnisse verarbeitet und

² Asynchronous JavaScript and XML

³ Information Fusion utilizing Instance Correspondences and Peer Mapping

archiviert. Zur Auswertung der gesammelten Daten werden *Analyseskripte* (siehe Abb. 2 rechts) definiert, deren Ergebnisse als XML-Daten verfügbar sind oder im Browser dargestellt werden können.

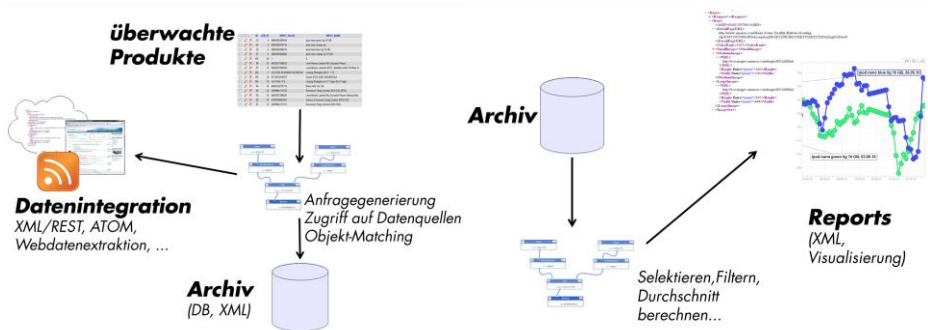


Abbildung 2: Generischer Workflow

Erstellen von Monitoring-Jobs: Je nach Typ des Extraktionsskripts werden unterschiedliche Anfragen an die *Monitoring-Datenquellen* erzeugt. Zum Erzielen guter Treffer in den *Monitoring-Datenquellen* sollten für jedes überwachte Produkt eindeutige Merkmale vorhanden sein. Diese können zum Beispiel die EAN⁴ oder ein eindeutiger Titel sein, der das Produkt von ähnlichen Produkten und Zubehörartikeln ausreichend abgrenzt. Dabei wird vom Benutzer nicht erwartet, dass er diese Produkteigenschaften genau kennt. Vielmehr wählt er nach Eingabe von Schlüsselwörtern aus Produkttiteln einen oder mehrere daraufhin generierte Vorschläge aus, die, falls verfügbar, Attribute wie EAN, Hersteller und Produktnamen enthalten.

Ausführung des Extraktionsskripts: In den *Extraktionsskripten* wird festgelegt, welche *Monitoring-Datenquellen* benutzt und wie Anfragen an diese Datenquellen erzeugt werden. Datenquellen können z.B. RSS-Feeds, Webservices oder per Screen-Scraping-Verfahren aufbereitete Webseiten sein. Je nach Abfragemöglichkeiten einer *Monitoring-Datenquelle* und vorhandenen Attributen der Eingabeobjekte ist die Strategie maßgebend, mit der Anfragen erzeugt werden, um alle relevanten Einträge in einer Datenquelle zu finden. Beim Verarbeiten großer Mengen von Eingabedaten kann die Reduktion der Anzahl gesendeter Anfragen entscheidend sein. Konfigurierbare Querygeneratoren dienen zum Erzeugen von Anfragen aus einer Liste von Eingabeobjekten. Bei Produkten können dafür im Normalfall Produkttitel, Metainformationen wie Hersteller oder EAN-Code verwendet werden. Die Ergebnismenge bei Anfragen, die aus dem Titel erzeugt werden, ist meist größer, beinhaltet aber oft Angebote für ähnliche Produkte oder Produktzubehör. Werden Anfragen mit Hilfe des EAN-Codes generiert, dann sind die Ergebnismengen in der Regel exakter aber von geringerer Kardinalität. Querygeneratoren ermöglichen es, bestimmte Attribute zu bevorzugen und auf andere Attribute auszuweichen, falls Attribute fehlen oder nicht durch Abfrageschnittstellen unterstützt werden. Datenquellen, welche die Anzahl erlaubter Anfragen limitieren oder

⁴ European Article Number - eindeutige Produktkennzeichnung für Handelsartikel

lange Antwortzeiten haben, benötigen Querygeneratoren, die die Anzahl der Anfragen reduzieren. Dazu können häufig vorkommende Schlüsselwörter oder OR-Verknüpfungen von Attributwerten unterschiedlicher Eingabeobjekte genutzt werden. Zur Identifikation der relevanten Treffer im Anfrageergebnis werden Objekt-Matching-Abläufe definiert, die eine String-Ähnlichkeit bestimmen. Zudem dienen diese Verfahren zum Entfernen von Duplikaten aus der Menge der gefundenen Angebote.

Auswertung: *Analyseskripte* greifen auf die archivierten Daten zu und sollen aggregierte Informationen bereitstellen, beispielsweise den Zeitverlauf des durchschnittlichen Preises oder die Händler mit den günstigsten Angeboten der vergangenen Tage. Das System bietet die Möglichkeit weitere *Analyseskripte* hinzuzufügen.

3 Einsatzszenarien

Für *Verbraucher* kann es relevant sein, schnell einen Überblick über die aktuellen Angebote zu erhalten, um daraus das günstigste auszuwählen. Das Monitoring von Produktpreisen kann ihn bei der Auswahl eines günstigen Kaufzeitpunkts unterstützen. Für *Online-Händler* ist es interessant, den aktuellen Marktpreis zu bestimmen, um z.B. Angebotspreise initial festzulegen. Eine Beobachtung über einen Zeitraum gibt Händlern die Möglichkeit, schnell auf Preisänderungen zu reagieren, den Angebotspreis nach oben oder nach unten zu korrigieren. Für *Hersteller* bietet sich eine Möglichkeit den aktuellen Marktpreis ihrer eigenen Produkte zu beobachten und sie in Relation zu Konkurrenzprodukten zu stellen. Zudem können Hersteller ermitteln, in welchen Shops und zu welchem Preis ihre Produkte angeboten werden.

4 Demonstration

Wir haben unsere Anwendung in den Domänen Produkt- und Flugpreisbeobachtung getestet. Teilnehmer der Vor-Ort-Demonstration können sich Ergebnisse bisher erfolgreicher Preisbeobachtungen ansehen. Dabei werden die verschiedenen Auswertungs- und Visualisierungsmöglichkeiten vorgeführt. Außerdem wird die Vorgehensweise bei der Erstellung von Monitoring-Jobs sowie die Abfrage und Aggregation aktueller Preisinformationen demonstriert.

Literaturverzeichnis

- [ETR09] Endrullis, S.; Thor, A. und Rahm, E.: Evaluation of Query Generators for Entity Search Engines. Proc. of Intl. Workshop on Using Search Engine Technology for Information Management (USETIM), 2009.
- [KH07] Klaus, P.; von Hören, Th.: Branchenführer Preisvergleichsportale. mpEXPERT, Grasberg, 2007.
- [Ra05] Rahm, E.; Thor, A.; Aumüller, D.; Do, H.H.; Golovin, N.; Kirsten, T.: Information Fusion utilizing Instance Correspondences and Peer Mapping. 8th International Workshop on the Web and Databases (WebDB), 2005.

ProCEM[®] Software Suite - Integrierte Ablaufsteuerung und -überwachung auf Basis von Open Source Systemen

Matthias Fischer, Marco Link, Nicole Zeise, Erich Ortner

Fachgebiet Entwicklung von Anwendungssystemen
Technische Universität Darmstadt
Hochschulstr. 1
D-64289 Darmstadt
<nachname>@winf.tu-darmstadt.de

Abstract: Zur Weiterentwicklung und Analyse von Unternehmen wird heutzutage die Ablauforganisation in den Mittelpunkt der Betrachtung gestellt. Diese Fokussierung wurde bereits im letzten Jahrhundert von Nordsieck, Kosiol, Gaitanides und anderen propagiert. Die IT-basierte Unterstützung der Unternehmensaktivitäten entwickelt sich analog, von sogenannten bereichsbezogenen „Silo-Systemen“ hin zu prozesszentrischen - Bereichs- ggf. auch Unternehmens-übergreifenden- Anwendungen. Die hier vorgestellte ProCEM[®] Software Suite folgt dieser Ausrichtung und erweitert den Funktionsumfang herkömmlicher Workflow Management Systeme u.a. um eine, über die Modellierung zu definierende, integrierte Überwachungskomponente. Das Gesamtsystem richtet sich speziell an den Bedürfnissen von kleinen und mittelständischen Unternehmen aus, die sowohl automatisierte, als auch manuelle (von Menschen durchzuführende) Prozessabläufe IT-basiert unterstützen möchten.

1 Allgemeine Informationen

Im Rahmen der vorgestellten Software Suite wird die von Prof. Dr. Erich Ortner etablierte ProCEM[®]-Methodik (Process-Centric Enterprise Modeling and Management) technologisch umgesetzt und erweitert (vgl. u.a. [Ort08]). Diese Methodik wurde auf Basis der in der Aufsatzreihe „Informatik als Grundbildung“ (vgl. [WOI04]) vorgestellten Schema-Management Theorie entwickelt. Im Kern setzt ProCEM[®] auf schematisierte Abläufe, die zur weiteren Spezifizierung, im Hinblick auf die tatsächliche Ausführung, auszustatten sind. Die geschieht zunächst ebenfalls auf Schemaebene, später jedoch gemäß dem Schema auch auf Ausprägungsebene (konkrete Ressourcen und Menschen werden referenziert). Betrachtet und rekonstruiert man Unternehmen aus ablauforganisatorischer Perspektive, so unterstützt die sogenannte „ProCEM[®] Software Suite“ in integrierter Form die wesentlichen Elemente des Prozesslebenszykluses. Die Prozessaufnahme, Unternehmens- und Anforderungsanalyse stellen stark manuelle Tätigkeiten dar, während das System die Strukturierung und Dokumentation unterstützt. Die anschließende Modellierung, unter Berücksichtigung verschiedenster Perspektiven (z. B. Prozess-, Organisations-, Datensicht, etc.), wird durch die entsprechenden Module der bereitgestell-

ten Suite unterstützt. Die aus diesen Modellen ableitbaren Prozessabläufe (spezifiziert um einige technische Elemente) dienen im Rahmen der späteren Ausführungsphase als Steuerungsparameter für die tatsächliche Ablaufsteuerung. Auf diese Weise lassen sich in geringer Entwicklungszeit prozesszentrische Anwendungen (Bestell- und Genehmigungsprozesse, standardisierte Abwicklungsverfahren, etc.) erstellen und im Unternehmen nutzbar machen.

Bereits im Rahmen der Modellierung können die Modelle dahingehend erweitert werden, dass Informationen für die Generierung aussagekräftiger Reports (Berichte) mit Business Intelligence zu Auswertungs- und Kontrollzwecken zur Verfügung stehen. Die Erstellung weiterer individueller Berichte ist darüber hinaus problemlos manuell möglich.

2 Gesamtarchitektur

Die ProCEM[®] Software Suite baut auf verschiedenen Open Source Basissystemen auf, wodurch eine stabile und erprobte Funktionsgrundlage gewährleistet werden kann. Durch die einheitlich gestaltete Oberfläche befindet sich der Endanwender (Client) in einem einzigen Programmkontext. Grundsätzlich wurde im Rahmen der Konzeption auf eine modulare Gesamtarchitektur, die eine möglichst effiziente Erweiterbarkeit und Austauschbarkeit ermöglichen soll, besonderen Wert gelegt. Die wesentlichen Modulbausteine können der Abbildung 1 entnommen werden. Aufgrund der gewählten Architektur ist das System sowohl auf lokalen als auch auf entfernten Systemen einsetzbar.

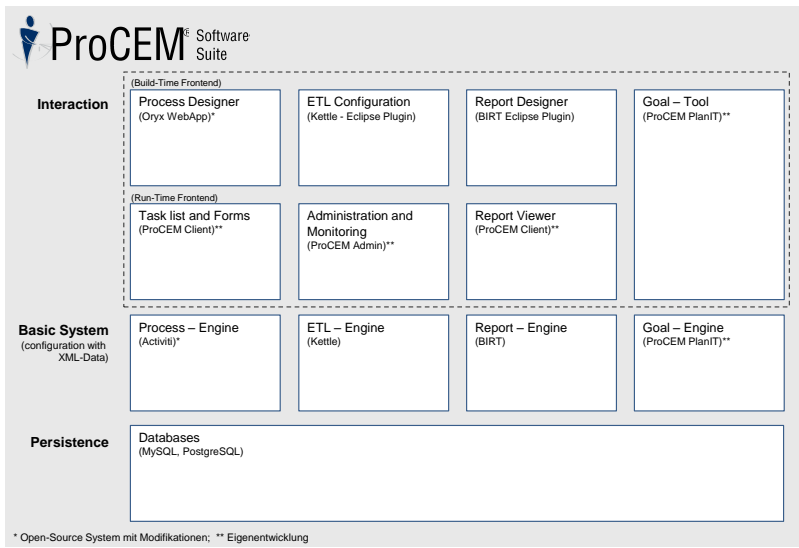


Abbildung 1: Architekturübersicht der ProCEM[®] Software Suite

Der Aufbau der Suite ist durch drei Ebenen charakterisiert, wobei die obere *Interaktions-ebene* in Build- und in Run-Time zu differenzieren ist. Das System arbeitet „interaktiv“ in der Hinsicht, dass sowohl die IT-bezogene als auch die Mensch-bezogene Informationsverarbeitung in die Gestaltung, Ausstattung und Durchführung der Prozesse mit eingehen. Insgesamt folgt das Gesamtkonzept einer Strukturierung auf Basis des Prozesslebenszykluses (vgl. hierzu auch [LO10]). Danach findet während der Build-Time die Schematisierung und Spezifizierung des Systemverhaltens statt. Zur Run-Time kommuniziert das System hauptsächlich mit den ausführenden und überwachenden Prozessbeteiligten. Bereits angesprochen wurden die zugrunde liegenden *Basissysteme*. Diese werden durch die erzeugten Schemata und die Oberflächen anwendungsbereit und stellen auch die Schnittstelle zur *Persistenzschicht* dar.

Auf die einzelnen Funktionsbereiche wird kurz innerhalb der Strukturierung: Modellierung, Ausführung und Überwachung eingegangen. Einige Module, wie z. B. die Benutzerverwaltung, etc. werden in Abbildung 1 und diesen Ausführungen nicht explizit erläutert.

Modellierung

Ausgangspunkt der Modellierung stellen die Prozessdiagramme im Standard BPMN 2.0 dar. Die vorgesehenen Prozessdetails können mit dem Oryx-Designer¹ webbasiert modelliert werden. Aufbauend aus Überlegungen aus [Zei10] wurden verschiedene Möglichkeiten implementiert, die bereits innerhalb der Modellierung die Voraussetzungen für eine automatische Berichtserstellung schaffen können. Weiterer Modellierungsaufwand entsteht im Hinblick auf die individuellen Berichte und den damit verbundenen ETL-Prozessen. Diese Modelle sind über die, auf unser System abgestimmten Eclipse Plugins BIRT² und Kettle³ vorzunehmen.

Ausführung

Wird die prozesszentrische Anwendung gestartet, so können die Benutzer, je nach Zugriffsrechten, im ProCEM Client Prozessinstanzen erzeugen, UserTasks abarbeiten oder technische wie fachliche Berichte überwachen. Auf fachlicher Ebene können z. B. Soll/Ist-Analysen durchgeführt werden, die basierend auf den operationalen und den ProCEM PlanIT Daten möglich sind. Als zentrale Steuerungskomponente wird die Workflow Engine Activiti⁴ verwendet.

Überwachung

Die Überwachung wird mit einer - speziell auf KMUs ausgerichteten - Meldekomponente (Teil von ProCEM PlanIT) sehr mächtig und zugleich einfach handhabbar. Diese Komponente greift auf ein Repositorium mit Metainformationen zu diversen Kennzahlen zu und kann diese verarbeiten. So können neben reinen Ist-Zuständen auch Planungsprozesse unterstützt werden. Spezielle Ampelreports zeigen an, ob Kennzahlenwerte in bestimmten Grenzbereichen liegen oder welche Zielerreichung (bzgl. der Planung) vorliegt. Diese Komponente ermöglicht es dem Benutzer ebenfalls, dynamische Reports zu generieren, so dass nur solche Werte angezeigt werden, die unternehmerisch kritisch sind.

¹ Vgl. <http://oryx-project.org/>

² Vgl. <http://www.eclipse.org/birt/>

³ Vgl. <http://kettle.pentaho.com/>

⁴ Vgl. <http://www.activiti.org/>

3 Zusammenfassung

Die ProCEM[®] Software Suite bietet eine einheitliche und integrierte Prozesssteuerung (Menschen via Tasklisten und Formulare, Systeme über technische Schnittstellen) und Prozessüberwachung. Hierbei sind neben generierten Prozess-bezogenen Berichten auch individuelle Reports möglich. Abbildung 2 zeigt zwei Bildschirmfotos des ProCEM Clients.

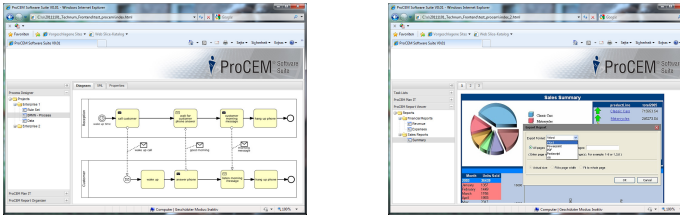


Abbildung 2: Bildschirmvorschau der Admin- (links) und Clientanwendung der Webapplikation

Ausblickend möchten wir das Gesamtsystem gerne mit weiteren, aus der Wissenschaft kommenden, Features ausstatten. Denkbar wäre z. B. die Überführung von Modellen in ein Logiksystem zur Untersuchung und Ableitung verschiedenster Eigenschaften und Abhängigkeiten [Fis10]. Weitere Ideen und Erweiterungen ergeben sich aber laufend aus den Anforderungen der Praxisprojekte und -einsätze. Hier wollen wir den KMUs eine Möglichkeit bieten, schnell und flexible Prozesse IT-technisch um- und einsetzen zu können. Im Rahmen der Demonstration soll allumfassend gezeigt werden, wie die Suite die Prozesse eines Modellunternehmens abbilden und die Mitarbeiter im täglichen Arbeitsalltag unterstützen kann.

Literatur

- [Fis10] Matthias Fischer. Eine Logik zur Abbildung arbeitsteiliger komplexer Prozessschemata. In Erich Ortner, Hrsg., *Konstruktive Informatik*, Seiten 23–30. 2010.
- [LO10] Marco Link und Erich Ortner. Dynamic Enterprise (as a composite Service System). In *International conference on service sciences*, Seiten 66–70. IEEE Computer Society, Los Alamitos CA, 2010.
- [Ort08] Erich Ortner. Process-centric Enterprise Modeling & Management (ProCEM). *Proceedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering*, 2008.
- [WOI04] Hartmut Wedekind, Erich Ortner und Rüdiger Inhetveen. Informatik als Grundbildung. 6 Aufsätze in: *Informatik-Spektrum* 27 (2004), Nr. 2, bis 28 (2005), Nr. 1, 2004.
- [Zei10] Nicole Zeise. Modellierung von Kennzahlen mit BPMN. In Gregor Engels, Markus Luckey und Wilhelm Schäfer, Hrsg., *Proceedings - Software Engineering 2010 - Workshopband (inkl. Doktorandensymposium)*, Jgg. 160 of *Lecture Notes in Informatics*, Seiten 63–74. Bonn, 2010.

Demonstration des Parallel Data Generation Framework

Tilman Rabl, Hatem Mousselly Sergieh, Michael Frank, Harald Kosch
Lehrstuhl für Verteilte Informationssysteme, Universität Passau
{rabl,mousselly,frank,kosch}@fim.uni-passau.de

Abstract: In vielen akademischen und wirtschaftlichen Anwendungen durchbrechen die Datenmengen die Petabyte Grenze. Dies stellt die Datenbankforschung vor neue Aufgaben und Forschungsfelder. Petabytes an Daten werden gewöhnlich in großen Clustern oder Clouds gespeichert. Auch wenn Clouds in den letzten Jahren sehr populär geworden sind, gibt es dennoch wenige Arbeiten zum Benchmarking von Anwendungen in Clouds. In diesem Beitrag stellen wir einen Datengenerator vor, der für die Generierung von Daten in Clouds entworfen wurde. Die Architektur des Generators ist auf einfache Erweiterbarkeit und Konfigurierbarkeit ausgelegt. Die wichtigste Eigenschaft ist die vollständige Parallelverarbeitung, die einen optimalen Speedup auf einer beliebigen Anzahl an Rechnerknoten erlaubt. Die Demonstration umfasst sowohl die Erstellung eines Schemas, als auch die Generierung mit verschiedenen Parallelisierungsgraden. Um Interessenten die Definition eigener Datenbanken zu ermöglichen, ist das Framework auch online verfügbar.

1 Einleitung

Cloudcomputing ist seit einigen Jahren ein reges Forschungsfeld. Das kontinuierliche Wachstum der Datenmengen, in vielen Anwendungen bis zu mehreren Petabytes, schafft neue Aufgaben für die Forschung. Um große Datenmengen zu verarbeiten werden automatisierte und adaptive Verfahren benötigt. In Rechnerverbunden mit tausenden von Knoten sind Hardwareausfälle keine Seltenheit, weswegen ein hohes Maß an Fehlertoleranz notwendig ist. In [RLH⁺09] haben wir einen Benchmark für adaptive Datenbanksysteme vorgestellt. In diesem Beitrag demonstrieren wir einen Datengenerator, der mit den Hauptzielen des Clustercomputing, Skalierbarkeit und Entkopplung, entworfen wurde.

Als Beispiel ist eine n-zu-n Beziehung zwischen zwei Relationen zu nennen. Um die Referenzen generieren zu können, müssen die existierenden Schlüssel der Relationen bekannt sein. Auf einem einzelnen Rechnerknoten ist es für gewöhnlich am schnellsten die teilnehmenden Relationen zu generieren und auszulesen um die Beziehung zu generieren. Wenn die Relationen aber über viele Rechner verteilt sind, ist es effizienter sie erneut zu generieren. Auf diese Weise kann die Beziehung vollständig unabhängig von den Basisrelationen generiert werden. Wenn für die Generierung verteilte Pseudozufallszahlengeneratoren verwendet werden, kann auch die Generation einzelner Relationen parallelisiert werden. Nachdem die Generierung deterministisch abläuft, können auch Referenzen unabhängig berechnet werden.

In [RFSK10] wurde das Parallel Data Generation Framework (PDGF) vorgestellt, das für Datengenerierung im Cloudmaßstab geeignet ist. PDGF ist hoch parallel und vollständig

konfigurierbar. Im Fokus der Implementierung standen Performanz und Erweiterbarkeit. Deshalb kann der Datengenerator auch leicht für andere Domänen eingesetzt werden. Die aktuelle Version verwendet keine Leseoperationen bei der Datengenerierung und reduziert damit die I/O und Netzwerk Last auf das absolute Minimum. Im folgenden Beitrag wird zunächst in Abschnitt 2 die Funktionsweise des Datengenerators kurz erläutert, dann in Abschnitt 3 ein Ausschnitt der Evaluierung gezeigt. Zuletzt beschreibt Abschnitt 4 die Demonstration, vor einer kurzen Zusammenfassung. PDGF ist online verfügbar¹.

2 Funktionsweise

Um einzelne Felder einer Tabelle unabhängig zu generieren, ohne teure Leseoperationen, werden jeder Spalte einer Datenbank ein Zufallszahlengenerator und ein Startwert (Seed) zugewiesen. Die verwendeten Zufallszahlengeneratoren sind hierarchisch angeordnet, wie in Abbildung 1 zu sehen. Auf diese Weise kann für jede Spalte ein deterministischer Seed erzeugt werden. Um einen einzelnen Wert in einer Spalte zu erzeugen, wird zunächst der Zufallszahlengenerator mit dem entsprechenden Seed gestartet. Dann wird die der Zeilennummer entsprechende Zufallszahl berechnet und aus dieser Zufallszahl mittels eines sogenannten Generators deterministisch der Wert erzeugt. Sollte eine einzelne Zufallszahl nicht ausreichend sein, kann die erzeugte Zufallszahl wiederum als Seed für einen Zufallszahlengenerator verwendet werden.

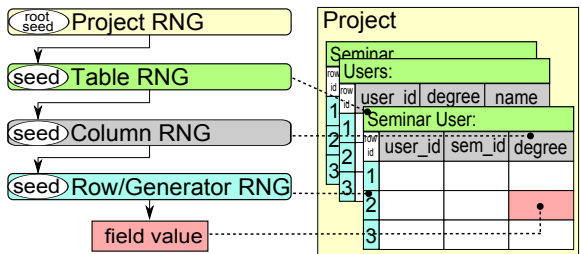


Abbildung 1: PDGFs Initialisierungsstrategie

Ein Zufallszahlengenerator wird als Startgenerator des Projekts verwendet, und dessen Startwert als Projektseed. Jeder Zufallszahlengenerator wird verwendet, um die Seeds für die in der Hierarchie nachstehenden Zufallszahlengeneratoren bzw. Generatoren zu erzeugen. So werden beispielsweise mit dem Startgenerator die Generatoren für die Tabellen geseedet. Nachdem es nur einen einzigen Projektseed gibt, können alle anderen Seeds davon abgeleitet werden und da die Anzahl der Tabellen und Spalten in einer Datenbank für gewöhnlich statisch sind, können alle Seeds im Speicher gehalten werden und müssen nur einmal bei der Initialisierung erzeugt werden. Daher muss der Datengenerator für gewöhnlich nicht die gesamte Hierarchie durchlaufen um einen Seed für einen Generator zu ermitteln. Es reicht aus den Zufallszahlengenerator erneut mit dem gespeicherten Seed zu initialisieren und zur entsprechende Zeilennummer zu springen. Danach kann die

¹PDGF Webseite - <http://www.fim.uni-passau.de/de/home/fakultaet/lehrstuehle/verteilte-informationssysteme/forschung/dbbench.html>

Zufallszahl an den Generator weitergegeben werden.

3 Evaluierung

Um die Geschwindigkeit und Skalierbarkeit des Generators zu testen wurden TPC-H Datenbanken generiert². Die Tests wurden auf einem Cluster mit 16 Knoten ausgeführt. Jeder Knoten hat zwei Intel Xeon QuadCore Prozessoren mit 2 GHz Taktfrequenz, 16 Giga-byte RAM und zwei 74 GB SATA Festplatten mit RAID 0 Konfiguration. Es wurden 2 Testreihen ausgeführt, zunächst wurde die Skalierbarkeit in Bezug auf die Datengröße getestet und danach die Skalierbarkeit in Bezug auf die Anzahl der teilnehmenden Knoten. Alle Tests zeigen, dass der Datengenerator in beiden Dimensionen linear skaliert. Die ausführlichen Testresultate können in [RFSK10] nachgelesen werden. An dieser Stelle wird nur nochmal der Vergleich mit dem in C implementierten dbgen gezeigt. Hierzu wurde mit PDGF die TPC-H Spezifizierung umgesetzt. TPC-H spezifiziert 8 Tabellen unterschiedlicher Größe und mit einer unterschiedlichen Anzahl an Spalten. Das Schema enthält Fremdschlüsselbeziehungen und verschiedenen Datentypen. Im Test wurde die Skalierbarkeit bezüglich der Datenmenge mit der des Standardgenerators von TPC-H - dbgen - verglichen. Dazu wurden mit beiden Generatoren Datenbanken der Größe 1, 10 und 100 GB generiert. Beide Generatoren wurden so gestartet, dass sie die 8 Kerne einer einzelnen Maschine voll ausnutzen konnten. In Abbildung 2 zeigt die Dauer der Generierung für beide Generatoren. Beide Skalen sind in logarithmischem Maßstab. Die Generierungsgeschwindigkeit beider Tools war durch die Prozessorgeschwindigkeit limitiert. Wie in der Abbildung zu erkennen ist, skaliert PDGF linear und hat eine vergleichbare Generierungsgeschwindigkeit wie eine spezialisierte C-Implementierung.

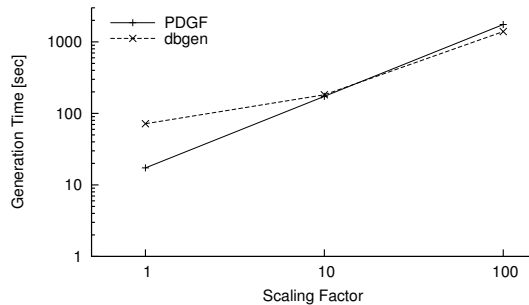


Abbildung 2: Vergleich der Generierungsgeschwindigkeit von dbgen und PDGF

4 Demonstration

Die Demonstration besteht aus zwei Teilen. Im ersten Teil wird der Datengenerator verwendet um eine einfache Datenbank zu erstellen. Im zweiten Teil wird ein einfaches "Feld-

²TPC-H Webseite - <http://www.tpc.org/tpch/>

GeneratorPlugin erstellt. Die Demonstration soll sowohl die einfache Anpassbarkeit, als auch Skalierbarkeit des Frameworks zeigen.

Generierung von Daten: Im ersten Teil der Demo wird ein einfaches Datenschema gezeigt und die entsprechende Datenbank generiert. Als erster Schritt wird eine XML Datei zur Schemabeschreibung erläutert und angepasst. Diese Datei enthält sowohl Parameter des Generierungsprozesses, wie die Datengröße oder das Ausgabeformat, als auch die Definition aller Schemaelemente und die entsprechenden Generierungsanweisungen. Für jede Tabelle wird aufgelistet welche Spalten existieren und entsprechende Generatoren und Verteilungen für die zu generierenden Daten spezifiziert. Nach der Spezifizierung werden Datenbanken mit verschiedenen Größen auf einer unterschiedlichen Anzahl an Rechenkernen generiert um die Skalierbarkeit zu demonstrieren.

Entwurf eines Feld-Generator Plug-In: Im zweiten Teil der Demonstration wird die einfache Erweiterbarkeit des Frameworks demonstriert, indem ein einfacher Generator erstellt wird. Die Schemadefinition wird um ein Feld, das diesen Generator verwendet erweitert und es wird erneut eine Datenbank generiert.

5 Zusammenfassung

In diesem Beitrag wurde das Parallel Data Generation Framework vorgestellt. Es ist einfach über XML Dateien anzupassen. Wie andere höher entwickelte Datengeneratoren (z.B. [SP04, HT07]) erlaubt es Abhängigkeiten zwischen Relationen und nicht-uniforme Verteilungen. Als Alleinstellungsmerkmal hat es allerdings ein neues Berechnungsmodell, das die deterministische Generierung von Zufallszahlen ausnutzt. Mit der Hilfe von Pseudozufallszahlengeneratoren können Abhängigkeiten in Datenbanken effizient aufgelöst werden, indem die referenzierten Werte erneut berechnet werden können. Die Evaluierung zeigt, dass eine generische Java-Implementierung des Modells eine äquivalente Generierungsgeschwindigkeit wie spezialisierte C-Implementierungen hat.

Literatur

- [HT07] Joseph E. Hoag und Craig W. Thompson. A Parallel General-Purpose Synthetic Data Generator. *SIGMOD Record*, 36(1):19–24, 2007.
- [RFSK10] Tilmann Rabl, Michael Frank, Hatem Mousselly Sergieh und Harald Kosch. A Data Generator for Cloud-Scale Benchmarking. In *TPC TC '10*, 2010.
- [RLH⁺09] Tilmann Rabl, Andreas Lang, Thomas Hackl, Bernhard Sick und Harald Kosch. Generating Shifting Workloads to Benchmark Adaptability in Relational Database Systems. *LNCS*, 5895(2009):116–131, 2009.
- [SP04] John M. Stephens und Meikel Poess. MUDD: a multi-dimensional data generator. In *WOSP '04*, Seiten 104–109, New York, NY, USA, 2004. ACM.

Measuring Energy Consumption of a Database Cluster

Volker Hudlet, Daniel Schall
AG DBIS, TU Kaiserslautern
67663 Kaiserslautern, Germany
{hudlet, schall}@cs.uni-kl.de

Abstract: Energy consumption of database servers is a growing concern for companies as it is a critical part of a data center's cost. To address the rising cost and the waste of energy, a new paradigm called *GreenIT* arose. Hardware and software developers are aiming at more energy-efficient systems. To improve the energy footprint of database servers, we developed a cluster of small-scale nodes, that can be dynamically powered dependent on the workload. This demo shows the measurement framework we set up to measure hardware components as well as an entire cluster of nodes. We'll exhibit the measurement devices for components and servers and show the system's behavior under varying workloads. Attendees will be able to adjust workloads and experience their impact on energy consumption.

1 Motivation

The growing energy consumption of data centers has become an area of research interest lately. Lots of effort is taken to improve the energy footprint of servers and to minimize overall energy consumption. Still, the amount of installed servers is constantly growing as more and more data is produced and has to be managed.

The focus on the servers' energy consumption is a rather young research field. Studies have shown that servers typically consume already more than 50 % of their peak energy when idle [THS10]. As the typical server load is between 20 to 50 % [BH07, Ran10], more energy-efficient approaches have to be developed to reduce the waste of energy of today's servers. This task addresses hardware developers as well as system designers and also the database community. Energy-proportional hardware (i.e., the energy consumed is proportional to the current load) would be highly desirable, unfortunately yet out of reach due to hardware limitations.

To avoid the limitations of a single large server in achieving energy proportionality, we are going to propose a new approach by employing commodity hardware to form an energy-proportional computing cluster. Small-scale server nodes (aka. *wimpy nodes* [AFK⁺09]) can be independently turned on and off, thus providing better scalability in terms of performance and power. By powering nodes based on the overall workload, the total energy consumption converges to an energy-proportional behavior. For now, however, standard server hardware components do not provide integrated probes to measure energy consumption of components or entire servers.

In this demo, we are going to introduce our idea of an energy-proportional database cluster and show how we measure power consumption. We will present an interactive front-end that controls the system’s workload and illustrates how the system reacts to rising and falling loads.

2 Architecture Overview

The key ideas of the database cluster and the architecture of the measurement framework are both outlined subsequently.

Database Nodes The database cluster consists of 10 identical nodes, two of them have attached 4 hard disks, these are considered as *data nodes*, the other 8 nodes are referred to as *compute nodes*. The compute nodes are interconnected by a 1Gb/s ethernet switch (*front-end switch*). Each node has 2 GB of DRAM and one Intel Atom CPU, a rather lightweight processor with low thermal design power. The Linux-based operating system of the nodes is booted from an attached USB stick. In order to fully support dynamic node powering, we are developing our own database management system, called *WattDB*. The system is capable of reacting to changing load situations and able to adapt the cluster to the workload. By switching nodes on and off, the overall energy consumption is made proportional to the load, thus energy proportional. Each node is providing feedback about its resource utilization to a coordinator, which uses this information to re-balance the cluster by automatically powering up and down nodes.

Energy Measurement We developed two devices for measuring energy consumption, one for measuring a single server down to the component level and a second device for measuring up to 10 servers in parallel.



(a) Photo
(b) Schema
Figure 1: Measurement Device for a Single Node

We are able to measure the energy consumption of a single node using a custom measurement device. Figure 1 shows a picture of the front of the device and a schematic plan.

The device is connected to a standard ATX power supply and able to measure voltage and current of each power line separately. That way, we are able to track the energy consumption of the mainboard and two connected hard disks. The measurements are converted to digital signals and can be read from a connected PC. We can track the power consumption with 1% accuracy at a 10 ms resolution. To sum up all corresponding power lines, we developed an evaluation tool that is able to perform mathematical operations on the measurement, e.g., integrate multiple sources over time to capture total power consumption. Additionally, the software is able to protocol the data to file.

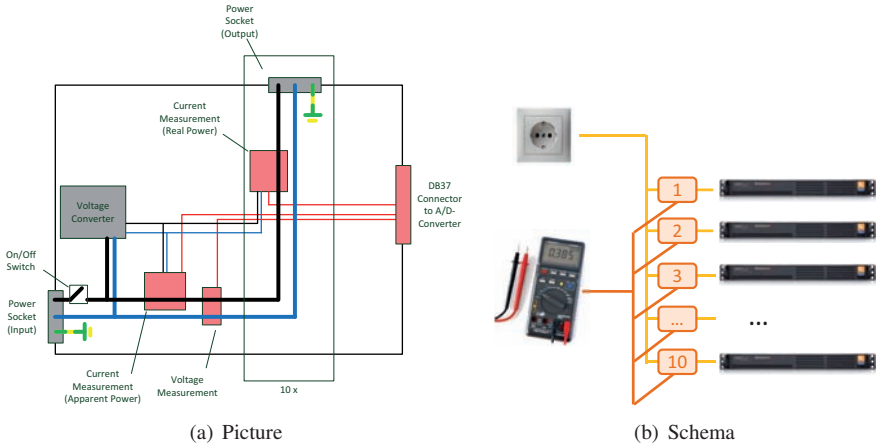


Figure 2: Measurement Device for a Node Cluster

In order to measure the energy consumption of a whole cluster of nodes, a similar approach is taken. Measuring such a cluster requires the overall energy consumption of each node – detailed power consumption of each components is not the main focus in this scenario. Therefore, a new measurement device is needed, as depicted in Figure 2(b). As illustrated, this device is able to track the power consumption of 10 server nodes independently. To measure current, *current transducers* are used. They provide high accuracy (less than 1% deviation) and do not influence the lines under measurement. An integrated circuit is used to provide the power factor of the entire cluster. The new device can measure the total effective power as well as the apparent power. Besides the current of each server, the input voltage is needed for calculating precise power consumption. Therefore, a separated circuit is introduced to measure the voltage. All obtained values are transferred in parallel over a bus to an outlet. The values can be read out using a connected A/D converter and a standard PC. The reporting interface is identical to the single-server tool, therefore the same reporting software can be used.

3 Demo Setup

To demonstrate the capabilities of our measurement device(s), a simple workload driver is used to trigger workload on the system. Changing energy consumption based on the differ-

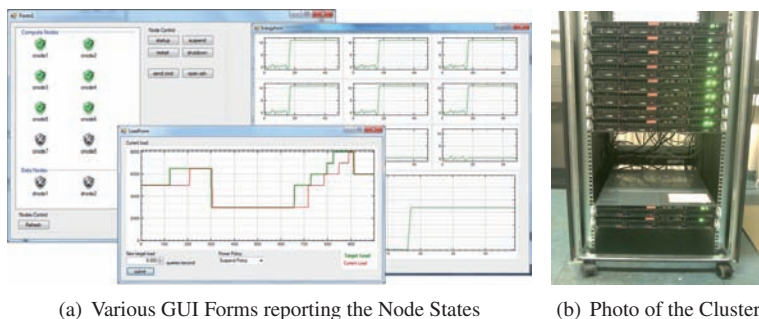


Figure 3: Demo Setup

ent loads is reported immediately by the devices. The power consumption of each node is visualized on a GUI, additionally, the cluster’s LEDs show the state of the nodes. The total power consumption is displayed in a separate graph. Figure 3(a) depicts a screenshot of the user interface for the cluster demo: The form on the left shows the various states (either *on*, *off* or *suspended*) of each of the nodes. On the right-hand side, the power consumption over time is plotted. At the front of Figure 3(a), a specific form makes the progress of the target and actual workloads visible. Both measurement devices are exhibited during the demo and can be seen in action.

The measurement device for a single server will have various storage disks attached, e.g. SSD and hard disk drives, and attendees can see the devices’ power consumption while triggering different kinds of workloads. Additionally, the measurement device for a server cluster will also be displayed, attached to server nodes and showing a reporting GUI as well. The user can alter the workload of the cluster using the corresponding GUI. The power policy for the cluster can also be changed. This policy controls the way nodes are powered up and down, what sleep states the nodes should enter and how aggressive the power management is done. The effect of the user’s actions is immediately visible – on the GUI as well as on the cluster itself (Figure 3(b)). Attendees can see how the system reacts to changing workloads and how the power consumption is affected.

References

- [AFK⁺09] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, pages 1–14, 2009.
- [BH07] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.
- [Ran10] Parthasarathy Ranganathan. Recipe for Efficiency: Principles of Power-Aware Computing. *Communications of the ACM*, 53(4):60–67, April 2010.
- [THS10] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah. Analyzing the Energy Efficiency of a Database Server. In *SIGMOD Conference*, pages 231–242, 2010.

***Snowfall*: Hardware Stream Analysis Made Easy**

Jens Teubner Louis Woods

ETH Zurich, Systems Group · Universitätstrasse 6 · 8092 Zurich, Switzerland

{jens.teubner | louis.woods}@inf.ethz.ch

1 Introduction

Field-programmable gate arrays (FPGAs) are chip devices that can be runtime-reconfigured to realize arbitrary processing tasks directly in hardware. Industrial products [Net, Xtr] as well as research prototypes [MTA09, MVB⁺09, SLS⁺10, TMA11] demonstrated how this capability can be exploited to build highly efficient processors for data warehousing, data mining, or stream analysis tasks.

On the flip side, the construction of dedicated hardware circuits requires considerable engineering efforts and skills that are often not available in application-focussed development teams. To bridge this gap, at ETH we have developed a set of tools that aid developers of high-performance stream processing solutions and enable agile hardware generation for changing application demands.

In this demonstration, we showcase *Snowfall*, a compiler tool for low-level stream analysis. Comparable to scanner generators for software-based systems (*e.g.*, *lex/flex*), *Snowfall* can be used to decode incoming data streams in hardware, react to low-level patterns in a stream, and perform initial input data analysis. *Snowfall* plays well together with *Glacier*, a query-to-hardware compiler that we described and demonstrated in [MTA09, MTA10]. A typical use case is to use *Snowfall* for input parsing and pre-processing, then perform SQL-style query processing on top with a hardware query plan obtained with the help of *Glacier*.

In the demo, we illustrate *Snowfall* based on a real-world use case with exceptionally high demands for throughput and latency. With the help of *Snowfall*, we perform *risk checking* for *financial trading* applications. *Snowfall* allows for a declarative description of the problem, yet will generate a hardware circuit that can process input streams in real time.

2 Field-Programmable Gate Arrays and State Machines

Field-programmable gate arrays (FPGAs) are programmable chip devices that can implement electronic circuits directly in hardware. They are programmed with a *hardware description language* such as VHDL or Verilog. Vendor-provided *synthesis tools* map circuit descriptions expressed in these languages to basic FPGA device primitives (*e.g.*, *lookup*


```

SOH                = 0x01;    # special value "SOH" (field delimiter)
FIXVersion          = "4.2";

# FIX Data Types
Length             = [0-9]+;
Qty                = [0-9]* ('.' [0-9]*)?;
String             = (any - SOH) *;

# FIX Fields
BeginString        = "8=FIX." FIXVersion SOH;
BodyLength          = "9=" Length SOH;
Checksum            = "10=" (any - SOH){3} SOH;
AnyField            = [1-9] [0-9]{0,3} "=" (any - SOH)* SOH;

NewOrderSingleMessage =
  BeginString BodyLength "35=D" SOH    # msg type = NewOrderSingleMessage
  AnyField * :>> "110=" Qty SOH        # quantity of executed order
  AnyField * :>> "55=" String SOH      # symbol
  AnyField * :>> "54=1" SOH            # this order is a buy
  AnyField * :>> CheckSum @check_order;

main := NewOrderSingleMessage;

```

Figure 1: Excerpt from a parser specification to decode FIX order messages.

tables; flip-flop registers; or *Block RAMs*). They generate a *bit stream* that, when uploaded to the FPGA, instantiates these primitives and realizes the hardware circuit.

Probably the most important design technique for FPGA circuits is the use of *finite state machines*, be it to implement the control logic that complements the data flow-oriented circuit components; to communicate with external devices; or to interpret data streams or protocols. Finite state machines fit the available FPGA chip resource types well and can run at very high speeds.

Designing the proper state machine for a given application need, however, can be tedious and error-prone. Even for relatively simple tasks, the necessary state machine can quickly grow too large to be truly understood by a human developer. And once programmed successfully, state machines tend to be hard to document and maintain. The problem is exacerbated by the necessity to express the state machine in VHDL or Verilog—languages that typical application developers are rarely familiar with.

3 *Snowfall*

Snowfall, which is part of a tool set that we develop in the context of the *Avalanche* project at ETH Zurich, addresses both aspects of the problem. It provides a high-level abstraction to express state machines and associated semantic actions.¹ *Snowfall* optimizes these state machines and emits VHDL code that implements them efficiently in hardware.

¹ *Snowfall* is based on the Ragel state machine compiler <http://www.complang.org/ragel/>.

Figure 1 shows an excerpt of the *Snowfall* code that decodes FIX messages for online trading applications. The code describes the lexical structure of buy orders (message type ‘NewOrderSingleMessage’ in the FIX specification) and inspects the quantity and stock symbol fields (FIX fields 110 and 55).

From the code in Figure 1, *Snowfall* will build a hardware state machine (expressed in VHDL) that recognizes the specified FIX message type. Whenever (parts of) the state machine have successfully matched on the input data, it will trigger the execution of *action code* blocks. These contain user-defined VHDL code that can be used to process lexical elements in the input stream (*lex/flex* are used in a similar way in software-based systems).

Since in this demo description we are restricted by space limits, Figure 1 shows only one example of how action code can be embedded into a *Snowfall* language specification. The `@check_order` annotation after the `Checksum` syntactical element specifies that the routine `check_order` should be invoked whenever a full `NewOrderSingleMessage` was successfully parsed.

A typical implementation of an action code like `check_order` will build an internal representation of a FIX order tuple (e.g., of schema `(quantity, symbol)`). The tuple is then forwarded on to further hardware logic that performs high-level analysis of the stream of FIX orders. One such analysis task could be an assessment of the *risk* associated with the orders made. For instance, we would like an *alert* to be raised whenever the order volume within a given time window exceeds a certain limit.

3.1 *Glacier*: A Query-to-Hardware Compiler

Higher-level stream analysis tasks are a good fit for *Glacier*, another part of our FPGA toolbox. *Glacier* is a SQL-to-hardware compiler. Given a query in an SQL dialect with streaming extensions, *Glacier* generates the VHDL description of a corresponding *hardware query plan*. The inner workings of *Glacier* are the subjects of [MTA09, MTA10].

To implement our risk analysis example, a *Glacier*-generated hardware plan consumes tuples that our FIX parser constructed in `check_orders` and performs *aggregation* and *windowing* on the tuple stream. For instance, the query

```
SELECT SUM (quantity) AS qsum
  FROM orders [ SIZE 600 ADVANCE 60 TIME ]
 GROUP BY symbol
```

aggregates all orders over a window of 10 minutes and reports the ordered quantities for each stock symbol every minute. A violation of risk limits could easily be detected from the aggregated output of this query; or a dedicated query could be written that only emits data in alert situations.

In summary, the combination of *Snowfall* and *Glacier* makes the development of stream processing solutions on FPGAs comparable to a typical software development work flow.

At the same time, generated solutions will run as bare-hardware implementations and thus benefit from the architectural advantages offered by the FPGA technology. In particular, the risk analysis example sketched here benefits from *network-speed processing*—no order will be missed even under peak load—and *real-time latency*—the system could react to risk violations within sub-microseconds time.

4 Demonstration Setup

The real value of our tool set results from the seamless interplay among our own tools, but also with commercial FPGA synthesis and simulation tools. To make this point, we will bring to Kaiserslautern not only *Snowfall*, but also a full FPGA design environment as well as FPGA hardware. We will show how a full example application can be developed, simulated, and debugged; and we will show how the resulting application can process a (synthetic) FIX message stream in real time.

Visitors of the demo will be invited to modify our code examples, write their own queries for *Glacier*, and inspect the generated hardware solution using commercial circuit visualization tools. The focus application for this demonstration, *Snowfall*, includes functionality to debug and visualize generated state machines. We will show and explain this functionality and illustrate how *Snowfall* eases the development of FPGA-based stream processing solutions.

Acknowledgements

This work was supported by SNF *Ambizione* grant number 126405 and by the Enterprise Computing Center (ECC) of ETH Zurich (<http://www.ecc.ethz.ch/>).

References

- [MTA09] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires—A Query compiler for FPGAs. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1), August 2009.
- [MTA10] Rene Mueller, Jens Teubner, and Gustavo Alonso. Glacier: A Query-to-Hardware Compiler. In *Proc. of the 2010 ACM SIGMOD Conference on Management of Data*, Indianapolis, IN, USA, June 2010.
- [MVB⁺09] Abhishek Mitra, Marcos R. Vieira, Petko Bakalov, Vassilis J. Tsotras, and Walid A. Najjar. Boosting XML Filtering Through a Scalable FPGA-Based Architecture. In *Int'l Conference on Innovative Data Research (CIDR)*, Asilomar, CA, January 2009.
- [Net] Netezza Inc. <http://www.netezza.com/>.
- [SLS⁺10] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. *Proceedings of the VLDB Endowment (PVLDB)*, 3(2), September 2010.
- [TMA11] Jens Teubner, Rene Mueller, and Gustavo Alonso. Frequent Item Computation on a Chip. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2011.
- [Xtr] XtremeData Inc. <http://www.xtremedata.com/>.

MOAW: An Agile Visual Modeling and Exploration Tool for Irregularly Structured Data

Horst Werner, Christof Bornhoevd
Robert Kubis, Hannes Voigt

SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf
horst.werner@sap.com

Abstract: The Mother of all Whiteboards (MOAW) is an innovative visual modeling and exploration tool for semi-structured information. It combines gesture-based user interaction with deep zooming, particle dynamics and the powerful data processing capabilities of SAP's newly developed Active Information Store. This application has been designed to convey complex information by easily created visual models which are backed by a formal representation and thus allow the fluid navigation to unlimited levels of detail and creation of different angles of view.

1 Background

In collaborative work, a common problem is the visual communication of complex situations, especially when large amounts of entities and relationships are involved. Common tools for visual communication are plain drawing boards (as e.g. embedded in online conferencing tools), slide presentation, diagramming or mind mapping applications.

Common to all these tools is the tradeoff between the effort needed to create the visualization and the expressive power of the model. For example, it is very easy to create visualizations in a plain drawing board or slide presentation application, but these visualizations usually only represent a very narrow view on the problem and need interpretation, which makes them easy to misunderstand. Often many different visualizations are created to illustrate different aspects of the same problem, but consistency between these cannot be ensured and navigation between them is one-dimensional (e.g. along a slide sequence).

Mind mapping and diagramming tools create more structured visualizations, but the price to pay is a much higher effort in the case of formal diagrams and a limited visual expressiveness in the case of mind mapping.

A new concept for visual modeling and user interaction has been developed which is implemented in the *Mother of all Whiteboards* (MOAW) prototype, based on the *Active Information Store* (AIS) [1] as the flexible data repository. The core of the prototype is the combination of modern user interaction paradigms like touch screens, gesture recognition, deep zooming and particle dynamics with the processing of semi-structured information, allowing the user to simultaneously create a visual model and a corresponding formal representation with very low effort.

The formal representation, consisting of *Info Items* with *Attributes* and *Associations*, ensures that the Visual Model is consistent and that the contents of the model can be processed with all the functionality we appreciate in databases: search, query, specific views, and analytics. However, the data model of the formal representation must correspond to the freedom of visual modeling offered by the Whiteboard-oriented UI. Hence, it must not confine the user to a predefined static schema, and nevertheless still convey a certain degree of semantics.

Therefore, the Active Information Store has been chosen as the underlying data persistence and processing layer. It provides a schema-flexible and scalable data management platform. Data can be stored in AIS without having to define a rigid database schema upfront. Rather, the schema of the data is derived automatically by the system based on the available data. Due to this flexibility data instances with highly irregular structure or heterogeneous data from different sources can be stored and processed efficiently. Most of the visual modeling and processing concepts map directly to corresponding concepts in the AIS data model and query language.

2 Applications

The MOAW has been developed to capture project-relevant information (e.g. processes, system landscapes etc.) in customer workshops and to display, analyze, annotate and enrich this information through all phases of a project. The whiteboard at the same time serves as a visual knowledge map and as a database that can be queried to generate new views on the existing content.

Typically, the whiteboarding process starts with simple boxes representing various facts and issues in a sticky note style. In the course of a session, the actual meaning of those boxes is clarified; they can be cast into tasks, topics, requirements, boundary conditions etc. These entities can be associated to each other and tagged with attributes of predefined or user-defined types. Details can be exposed by zooming in and creating new structures within the boxes.

The objects created on the whiteboard (including associations and attributes) are stored together with master data about people, skills, products, projects in a common knowledge base. The mass data operations offered by the AIS, which are exposed by very simple, gesture-based interaction patterns, allow the quick exploration and generation of multiple views on all this content.

By blurring the borderlines between visual modeling, formal modeling and data exploration, the demonstrated technology also shows new ways for efficient user interaction in many other knowledge-intensive application areas, e.g. the collaborative discovery of trends from publicly available information and fraud detection.

3 Basic Interaction Types

Almost all user interaction in the MOAW is based on:

- the ability to move and zoom in and out of the viewport very quickly,
- dragging of objects and
- drawing of gestures as temporary lines on the screen in order to interact with the visual content.

The fluent deep zooming in combination with a Level of Detail (LOD) mechanism allows the user to navigate from a high-level overview to the finest details in seconds and without any hard context changes.

Drag and drop operations are used to move visual objects into and out of sets and the drawn gestures are an extremely fast and yet intuitive way of invoking more complex interactions like creating objects and object sets and of performing queries.

3.1 Creation of Objects and Associations

Objects are created by simply drawing a contour on the screen, where the type of the contour corresponds to the type of object. By typing the caption of a newly created visual object, the user determines whether a *new* data object (Info Item) is created or whether the visual object is mapped to an *existing* Info Item in the AIS. In the latter case all the properties (attributes and associations with other Info Items) of the Info Item can be shown in a property sheet and selectively integrated into the visual model. New properties of an Info Item can be added or removed flexibly at any point in time.

Associations between Info Items can be created by drawing a line between the corresponding visual objects. The type of the association is inferred from metadata attached to the Info Item types. If no such metadata exists, the item types themselves are used as association types. These can be changed at any time in the property sheet.

3.2 Mass Manipulation of Objects

Sets of objects can be selected by drawing an ellipse around them or created by using a gesture-based query mechanism. These *visual item sets* correspond to the concept of Info Item Sets in AIS and for every operation the AIS offers for such sets, a corresponding gesture is implemented:

- A set of common properties for all set elements can be defined (e.g. an association with a certain person as displayed in Fig 1). Consequently, moving visual items into or out of the set's contour will change the corresponding Info Item's properties in the AIS.

- A query is performed by drawing a line from the outside to the inside of a set with specified common properties. All Info Items matching these properties will be retrieved from the Active Information Store, and corresponding visual items will be generated in the visual item set if they are not already there.
- The AIS offers the ability to derive the common properties for a given set of Info Items. Correspondingly, opening the property sheet for a selection of objects will invoke this function and display the extracted properties.

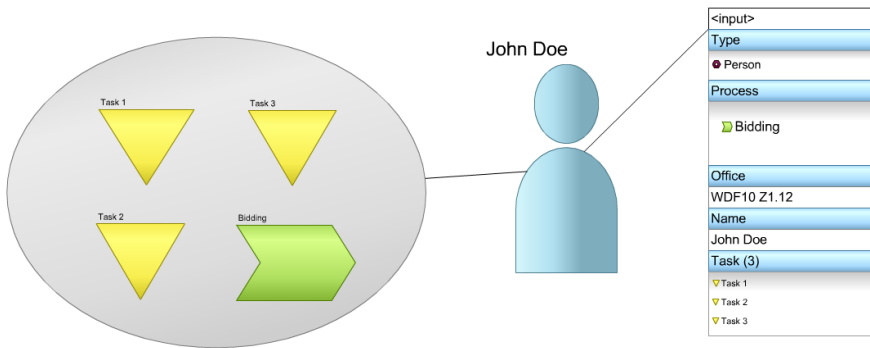


Fig 1: Association of multiple objects to a person and their reflection in a property sheet.

4 Demo

The demo will show the visual modeling of a consulting project with the MOAW and explain how the corresponding Info Items are created, maintained and processed in AIS.

Starting with an empty screen, visual objects for the project itself and several sub-topics, process steps, tasks, people etc. will be created by gestures and then partly bound to existing Info Items by means of an autosuggest feature.

The association of single tasks with process steps and people by gestures and the specification of further properties in a property sheet will be shown. Then the selection and association of whole sets of tasks with people will be demonstrated, as well as the subsequent maintenance of priorities for sets of tasks.

Then, a set of people will be selected, and the set of common properties (skills, location) will be extracted by a gesture. A subsequent query gesture will find additional people with the same properties, some of which are then added to the project by drag and drop.

References

[1] "Active Information Store – A Repository for Flexible Information Management, Exploration, and Analysis", C. Bornhoevd et al., SIGMOD 2011, submitted

Touch it, Mine it, View it, Shape it

Martin Hahmann, Dirk Habich, Wolfgang Lehner

TU Dresden; Database Technology Group; Dresden, Germany

Abstract: To benefit from the large amounts of data, gathered in more and more application domains, analysis techniques like clustering have become a necessity. As their application expands, a lot of unacquainted users come into contact with these techniques. Unfortunately, most clustering approaches are complex and/or scenario specific, which makes clustering a challenging domain to access. In this demonstration, we want to present a clustering process, that can be used in a hands-on way.

1 Introduction

Clustering is the partitioning of a set of objects into clusters [EK SX96, JMF99], so that similar objects are in the same cluster, while dissimilar objects are not. In order to create a clustering, an appropriate algorithm must be selected, parameterized and executed. The obtained result is evaluated and if necessary, algorithm and/or parameters are modified and the clustering is generated again. Each of these actions determines the course and outcome of the clustering process [JL05, JMF99]. Even so, user-support is lacking in practise, which made ‘trial and error’ a common approach to clustering, for users not familiar with the subject. Obviously, this often results in numerous iterations, unsatisfying results and eventually user frustration.

With *ensemble clustering* [GMT05, SG02], an alternative to single-algorithm clustering has been established. This approach creates multiple partitionings of a data set—the cluster ensemble—and aggregates them into one final clustering result. In doing so, quality and robustness are increased in comparison with single input clustering [GMT05, SG02]. Additionally, this procedure eases algorithm selection and parameterization. However, the overall resemblance to ‘trial and error’ remains, as unsatisfying aggregation results can only be adjusted by modifying the cluster-ensemble and repeating its creation and aggregation.

In our previous work, we have already addressed some of the described issues. In [HVR⁺09] we proposed an extended aggregation algorithm, utilizing soft clustering input and allowing result adjustments by parameterization of the aggregation only. To enable user support, we introduced an interactive visualization to control our aggregation, assist with result interpretation and indicate appropriate result adjustments [HHL10b].

In our demonstration, we present a clustering process composed from this components and show how this easy-applicable process allows the step-by-step refinement of a clustering.

2 Process

In this section, we will outline the structure and components of our clustering process. This process incorporates an algorithmic platform, which covers selection and execution of algorithms and a visual-interactive interface, assisting the user during result evaluation and modification.

The already introduced ensemble-clustering concept, built the conceptual starting point in the development of our algorithmic platform. Besides positive effects on the clustering result, this method aids the user by reducing the emphasis on the identification of a single optimal algorithm/parameter combination. All existing aggregation approaches we examined, lacked controllability [HVR⁺09], thus result adjustments were only possible through modification of the input clusterings. Unfortunately this effectively nullifies the benefits regarding user support, since now a whole set of clusterings must be reconfigured. To overcome this issue, we proposed our enhanced *flexible clustering aggregation* concept [HVR⁺09], which extends the classic approach in three major areas. First, the aggregation input is enriched with additional information about object-cluster relations, by utilizing *soft clustering* algorithms [Bez81] to generate the cluster ensemble. To benefit from this gain in information, the core aggregation method was modified in a second expansion. Finally, these arrangements allowed the derivation of a scoring function and with it the implementation of a control mechanism for the clustering aggregation. With *flexible clustering aggregation* users can adjust results without touching the cluster-ensemble. The necessary parameters could be abstracted in a user-friendly way, so that clusterings are adjusted by “*merging*” and “*splitting*” clusters.

To support result interpretation and identification of appropriate adjustments, we developed a visualization concept that is tightly coupled to our algorithmic platform. Our approach *augur* [HHL10b] can be seen as a hybrid between the two major groups of data/clustering visualizations, which are: (i) data-driven and (ii) result-driven. The first group depicts all objects and dimensions of the data, resulting in incomprehensible presentations and information-overload, as datasets exceed a certain scale. In contrast, the second group is relatively scale-invariant since only analysis results are presented (e.g. a clustering can be depicted as bar chart showing relative cluster sizes). Unfortunately these visualizations often shows not enough information. The hybrid character of our approach is achieved by visualizing the result and its relations to data, which are already incorporated in the soft input of our aggregation.

In compliance with Shneiderman’s mantra, ‘*overview first, zoom and filter, then details-on-demand*’ [Shn96], our visualization features views for theses three levels of detail. Our overview acts as a visual entry point and shows basic characteristics of the clustering aggregate, like relative size and the distances between the prototypes (centroids) of all clusters. If the user identifies clusters of interest in the overview, e.g. two very close clusters, these can be selected individually to get more information regarding their composition and their relations to other clusters, thus performing ‘*zoom and filter*’. More detailed information concerning a cluster’s internal similarity resp. composition are presented in the attribute view.



Figure 1: Touch it, Mine it, View it, Shape it

By combining the *flexible clustering aggregation* with our *augur* visualization we, devised our clustering process. The course of the process begins with the presentation of an initial clustering result to the user. Using *augur*, this result is interpreted and the parts that need adjustment are identified. Via an interactive component of the visualization, the users modifications are forwarded to the algorithmic platform. After they are applied to the clustering, the adjusted result is again presented using the *augur* visualization. With this procedure, users can refine clustering results in an iterative manner. A theoretic description of this process model and its components, e.g. the available user-feedback operations, are published in [HHL10a].

3 Demo Details

The demo at BTW comprises a detailed explanation of the necessary concepts and components of our process and its live demonstration. We are going to show how our visualization and interaction concepts can be used to conduct a visually-driven exploration of scientific data sets. Furthermore, we will prepare some application scenarios based on synthetic as well as real-world data-sets. Within these scenarios, we will illustrate the benefits of our iterative refinement approach with regard to its handling by users not familiar with the domain of clustering. Additionally, we want to use the BTW environment to discuss possible future developments for our employed aggregation algorithms and visualization concept with interested demo visitors.

4 Summary

In this paper, we introduced our hands-on clustering process, which offers inexperienced users an accessible way to generate a satisfying clustering. Execution and Parameterization are eased by the user-friendly character of our algorithmic platform. In tight coupling with this platform, our visual-interactive user-interface, supports the interpretation of clustering results by revealing characteristics of clusters as well as relations between them and the underlying data. This result- and relation-oriented approach offers assistance to the user during the identification of appropriate result modifications. In contrast to existing clustering procedures our approach allows the iterative refinement of a clustering.

References

- [Bez81] James C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms*. New York: Plenum, 1981.
- [EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. of KDD*, 1996.
- [GMT05] Aristides Gionis, Heikki Mannila, and Panayiotis Tsaparas. Clustering Aggregation. In *Proc. of ICDE*, 2005.
- [HHL10a] Martin Hahmann, Dirk Habich, and Wolfgang Lehner. Evolving Ensemble-Clustering to a Feedback-Driven Process. In *Proceedings of the IEEE ICDM Workshop on Visual Analytics and Knowledge Discovery (VAKD)*, 2010.
- [HHL10b] Martin Hahmann, Dirk Habich, and Wolfgang Lehner. Visual Decision Support for Ensemble-Clustering. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM)*, 2010. (to appear).
- [HVR⁺09] Martin Hahmann, Peter Volk, Frank Rosenthal, Dirk Habich, and Wolfgang Lehner. How to Control Clustering Results? Flexible Clustering Aggregation. In *Advances in Intelligent Data Analysis VIII*, pages 59–70, 2009.
- [JL05] Anil Jain and Martin Law. Data Clustering: A Users Dilemma. *Pattern Recognition and Machine Intelligence*, pages 1–10, 2005.
- [JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3), 1999.
- [SG02] Alexander Strehl and Joydeep Ghosh. Cluster Ensembles — A Knowledge Reuse Framework for Combining Multiple Partitions. *Journal of Machine Learning Research*, 3, 2002.
- [Shn96] Ben Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society.

Metadata-driven Data Migration for SAP Projects

Martin Oberhofer, Albert Maier, Thomas Schwarz, Manfred Vodegel

IBM Germany - Research and Development GmbH
Schoenaicherstrasse 220
71032 Boeblingen, Germany
{martino, amaier, schwarz, vodegel}@de.ibm.com

Abstract: SAP applications are mission-critical for many enterprises today. However, projects to introduce a new SAP solution or consolidate existing SAP solutions often fail respectively overrun budget and time. A common root cause is the underestimation of data migration work. Data quality in legacy systems is often not sufficient for SAP, and specifications of the target data model often change very late in the project lifecycle, e.g. due to new business requirements or new insights about legacy systems and legacy business processes. This can cause significant re-work in the ETL jobs that extract data from source systems, cleanse that data and load it into the target SAP system(s). We apply a model-driven architecture (MDA) approach [MP10] to such data migration projects. We generate ETL infrastructure from SAP metadata. This novel approach (known as the IBM Ready-To-Launch (RTL) for SAP solution [Ibm10]) significantly reduces project risk and cost. In addition, data quality is addressed and improved. Our demo will show programmatic access to SAP metadata and its systematic exploitation throughout the data migration project, including the generation of logical and physical data models from this metadata, and the generation of ETL jobs.

1 The RTL for SAP Solution – System Architecture

The IBM InfoSphere Information Server platform is the technology foundation of the RTL for SAP solution. It delivers enterprise information integration capabilities across all integration areas such as discovery, data profiling, ETL, replication, federation [GH10] and SOA services. It also offers SAP-certified application connectors to extract data from SAP applications and load data into SAP applications. By applying an MDA approach, RTL provides several novel capabilities to SAP data migration projects. On the one hand, using MDA principles, functional data requirement specifications are linked to business process specifications. On the other hand, functional data requirement specifications are linked to programmatic SAP metadata access. RTL has a so-called Rapid Generator component which can generate ETL jobs for data exchange with SAP systems. By linking ETL jobs model-based to functional data requirement specifications, which in turn are linked to business process specifications, it is now easy to adapt to changes in the business process requirements. If, for example, a change of a business process results in an additional, new business object attribute, the ETL jobs just need to be regenerated whenever such a change happens. In a traditional approach [LN06] such changes are often not detected until system integration test and require manual, often cumbersome adjustments in the ETL code base and additional testing.

Trademarks: IBM logos are trademarks or registered trademarks of the International Business Machines Corporation in the United States, other countries, or both. SAP Netweaver, SAP ERP, SAP R/3, SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

Using the conceptual system architecture shown in Figure 1, we will now explain where we apply the Rapid Generator capability. On the source system side, there might be some legacy SAP R/3 systems as well as some non-SAP systems. The target(s) are SAP Netweaver systems such as SAP ERP. While the data is moved from source to target, there are three distinct areas where the data is persisted while in transit – the areas are usually separated using database schemas within the same database. The staging (STG) area is modelled identically to the source system data models. For SAP sources, the STG area is modelled and the tables are automatically physically created exploiting programmatic access to SAP metadata. Based on the same metadata Rapid Generator generates all extract jobs extracting the data from SAP into STG. The alignment (ALG) area is programmatically created from the SAP target system data model. It is not an exact copy of the SAP target data model. For example, there might be a need to map a hierarchical IDoc model to a relational model or to not enforce foreign key relationships yet. The rationale is that we want to be able to get all records from all sources into ALG. Once the data from all sources is in ALG in a common format, we can analyze, report, and later resolve all data quality issues on it in a uniform way.

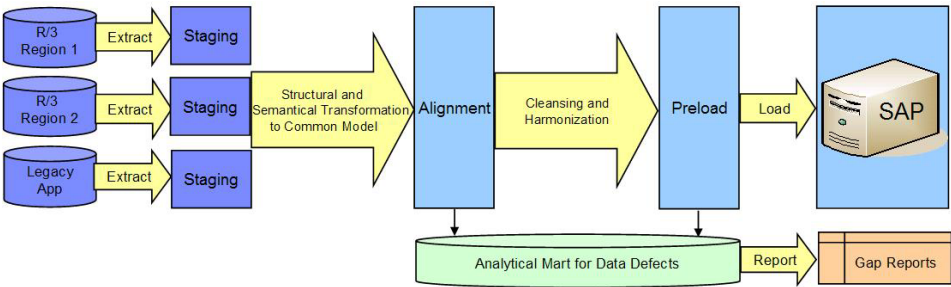


Figure 1: Conceptual System Architecture of the RTL for SAP Solution

Two steps are done when data is moved from STG to ALG: Structural alignment which means harmonizing the various source data models into a common model and semantical alignment which means transcoding the lookup values from the source system into the lookup values understood by the SAP target system. The country code value for Germany in a source system might be for example **62** whereas the corresponding code value in the SAP target system might be **DE**. Once the data is in ALG, data cleansing is performed. Common cleansing operations include data standardization, matching and de-duplication. The third area known as preload (PLD) area is also generated from the data model of the SAP target system. Data types and foreign key relationships are enforced now. Moving data from ALG to PLD requires a structural transformation generated by Rapid Generator. All jobs required to load the data from PLD into SAP are also generated by Rapid Generator. In addition, Rapid Generator generates all jobs to extract the data from all lookup tables used for attributes in the model. The values of the lookup tables are needed for the Data Validity Gap Report (DVGR) which measures how many of the source records have code values that are not valid in the context of the SAP target. The DVGR is just one type of gap reports in RTL. Other gap reports are checking data completeness, data validity and field length. In general, gap reports are used to measure data quality defects while the data resides in ALG, and to measure data

load readiness in PLD. They are executed periodically during the data migration project to provide the project manager the ability to see how much progress has been made since their last execution. All gap reports are dynamically created. Technically they are based on SQL templates and take SAP metadata into account. For example, if in the SAP target system an optional attribute becomes mandatory (a normal SAP customization step), this change is visible in the SAP metadata, and the Data Completeness Gap Report would then start to report how many source records are not providing values for this attribute. Technically, SQL statements are used for these checks.

2 Demo Scenario

The key ingredient for the RTL for SAP solution is SAP metadata. Without a programmatic access to SAP metadata, the modelling of the STG, ALG and PLD areas for SAP source and target systems is cumbersome, very time consuming and, if done manually, prone to errors. An SAP Netweaver ERP solution has about 70000 logical tables – some of them are data tables where data tables can have more than 150 attributes. Dozens of these attributes could be backed by lookup tables. In addition, without an understanding of the dependencies among SAP business objects and their data models, it is difficult to define the right order of migrating data for the various SAP business objects. For example, transactional business objects like orders require that master data business objects such as product and customer have been successfully loaded. Furthermore, SAP exposes different attribute subsets for the business objects through different application-level load interfaces such as IDoc. An IDoc is a hierarchical structure representing a business object. It could easily have more than 300 attributes. Without an in-depth understanding of the technical metadata of these interfaces, and how they are related to the logical data model of the SAP business objects, it is difficult to prepare the data for load. Combining this with the fact that the functional data specifications might change while the data migration project is executed drives the need to automate the loading with a model-driven approach.

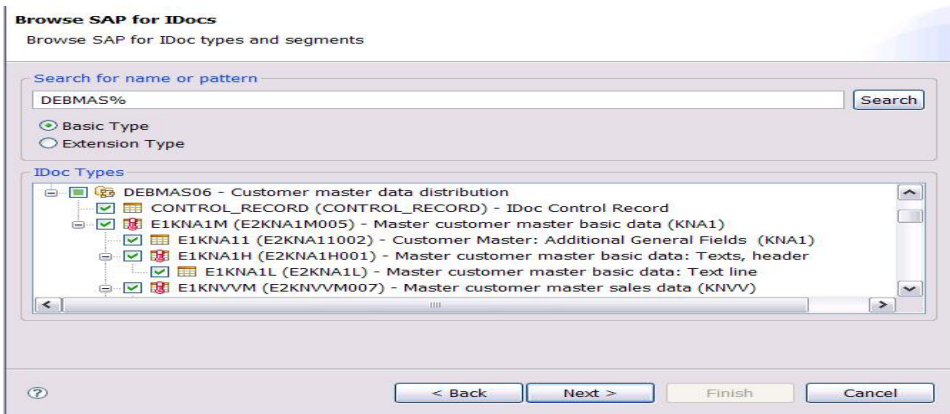


Figure 2: IBM InfoSphere Rapid Generator for SAP applications – IDoc example

The Rapid Generator approach delivers many benefits. Re-generating the physical tables for STG, ALG and PLD is seamless because the programmatic SAP metadata access reacts to data model changes in the SAP systems. Rapid Generator removes the manual efforts for extract and load jobs for SAP systems almost entirely. It automates the generation of jobs extracting a large numbers of lookup tables as well as the related lookup description tables by automatically joining them together and making them available in ALG for code value transcodings and gap reports. If the IDoc interface is used for load, the technical details in linking together all the tables at the right place in the hierarchy with the right foreign key relationships are now hidden from the ETL developer. This removes a previously tricky and error prone manual task. An example for the customer business object for SAP ERP with the corresponding DEBMAS IDoc structure is shown for a portion of the tree in Figure 2.

Among others, in the demo we will show the following steps: We demonstrate how to access SAP metadata programmatically, build corresponding logical and physical models, and persist SAP metadata in an auxiliary schema in the RTL database. Also, the functional data requirement specifications are linked to and persisted in this auxiliary schema. Then we demonstrate how the STG, ALG and PLD areas are instantiated in a model-driven way. Third, we show how Rapid Generator consumes the models and generates jobs. An example load job using the IDoc interface is shown in Figure 3.

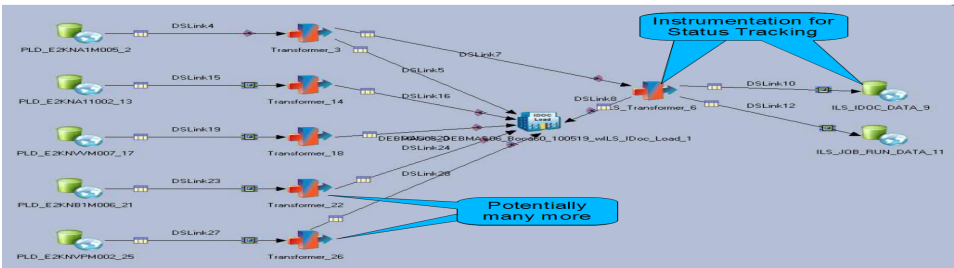


Figure 3: IBM InfoSphere Rapid Generator for SAP applications – IDoc example

References

- [GH10] Godinez, M., Hechler, E., Koenig, K., Lockwood, S., Oberhofer, M., Schroeck, M.: The Art of Enterprise Information Architecture – A Systems-Based Approach for Unlocking Business Insight. Pearson, 1st Edition, 2010.
- [Ibm10] IBM whitepaper: Driving successful business transformation: persistent organizational excellence with IBM Ready-to-Launch for SAP, October 2010, <ftp://public.dhe.ibm.com/common/ssi/ecm/en/imw14493usen/IMW14493USEN.PDF>.
- [LN06] Leser, U., Naumann, F.: Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen. dpunkt Verlag, 1st Edition, 2006.
- [MP10] Molina, J.-C., Pastor, O.: Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling. Springer, 1st Edition, 2010.

GI-Edition Lecture Notes in Informatics

- P-1 Gregor Engels, Andreas Oberweis, Albert Zündorf (Hrsg.): Modellierung 2001.
- P-2 Mikhail Godlevsky, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications, ISTA'2001.
- P-3 Ana M. Moreno, Reind P. van de Riet (Hrsg.): Applications of Natural Language to Information Systems, NLDB'2001.
- P-4 H. Wörn, J. Mühling, C. Vahl, H.-P. Meinzer (Hrsg.): Rechner- und sensor-gestützte Chirurgie; Workshop des SFB 414.
- P-5 Andy Schürr (Hg.): OMER – Object-Oriented Modeling of Embedded Real-Time Systems.
- P-6 Hans-Jürgen Appelrath, Rolf Beyer, Uwe Marquardt, Heinrich C. Mayr, Claudia Steinberger (Hrsg.): Unternehmen Hochschule, UH'2001.
- P-7 Andy Evans, Robert France, Ana Moreira, Bernhard Rumpe (Hrsg.): Practical UML-Based Rigorous Development Methods – Countering or Integrating the extremists, pUML'2001.
- P-8 Reinhard Keil-Slawik, Johannes Magenheimer (Hrsg.): Informatikunterricht und Medienbildung, INFOS'2001.
- P-9 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Innovative Anwendungen in Kommunikationsnetzen, 15. DFN Arbeitstagung.
- P-10 Mirjam Minor, Steffen Staab (Hrsg.): 1st German Workshop on Experience Management: Sharing Experiences about the Sharing Experience.
- P-11 Michael Weber, Frank Kargl (Hrsg.): Mobile Ad-Hoc Netzwerke, WMAN 2002.
- P-12 Martin Glinz, Günther Müller-Luschnat (Hrsg.): Modellierung 2002.
- P-13 Jan von Knop, Peter Schirmbacher and Viljan Mahni_ (Hrsg.): The Changing Universities – The Role of Technology.
- P-14 Robert Tolksdorf, Rainer Eckstein (Hrsg.): XML-Technologien für das Semantic Web – XSW 2002.
- P-15 Hans-Bernd Bludau, Andreas Koop (Hrsg.): Mobile Computing in Medicine.
- P-16 J. Felix Hampe, Gerhard Schwabe (Hrsg.): Mobile and Collaborative Business 2002.
- P-17 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Zukunft der Netze –Die Verletzbarkeit meistern, 16. DFN Arbeitstagung.
- P-18 Elmar J. Sinz, Markus Plaha (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2002.
- P-19 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3.Okt. 2002 in Dortmund.
- P-20 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3.Okt. 2002 in Dortmund (Ergänzungsband).
- P-21 Jörg Desel, Mathias Weske (Hrsg.): Promise 2002: Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen.
- P-22 Sigrid Schubert, Johannes Magenheimer, Peter Hubwieser, Torsten Brinda (Hrsg.): Forschungsbeiträge zur "Didaktik der Informatik" – Theorie, Praxis, Evaluation.
- P-23 Thorsten Spitta, Jens Borchers, Harry M. Sneed (Hrsg.): Software Management 2002 – Fortschritt durch Beständigkeit
- P-24 Rainer Eckstein, Robert Tolksdorf (Hrsg.): XMIDX 2003 – XML-Technologien für Middleware – Middleware für XML-Anwendungen
- P-25 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Commerce – Anwendungen und Perspektiven – 3. Workshop Mobile Commerce, Universität Augsburg, 04.02.2003
- P-26 Gerhard Weikum, Harald Schöning, Erhard Rahm (Hrsg.): BTW 2003: Datenbanksysteme für Business, Technologie und Web
- P-27 Michael Kroll, Hans-Gerd Lipinski, Kay Melzer (Hrsg.): Mobiles Computing in der Medizin
- P-28 Ulrich Reimer, Andreas Abecker, Steffen Staab, Gerd Stumme (Hrsg.): WM 2003: Professionelles Wissensmanagement – Erfahrungen und Visionen
- P-29 Antje Düsterhöft, Bernhard Thalheim (Eds.): NLDB'2003: Natural Language Processing and Information Systems
- P-30 Mikhail Godlevsky, Stephen Liddle, Heinrich C. Mayr (Eds.): Information Systems Technology and its Applications
- P-31 Arslan Brömme, Christoph Busch (Eds.): BIOSIG 2003: Biometrics and Electronic Signatures

- P-32 Peter Hubwieser (Hrsg.): Informatische Fachkonzepte im Unterricht – INFOS 2003
- P-33 Andreas Geyer-Schulz, Alfred Taudes (Hrsg.): Informationswirtschaft: Ein Sektor mit Zukunft
- P-34 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenber, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 1)
- P-35 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenber, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 2)
- P-36 Rüdiger Grimm, Hubert B. Keller, Kai Rannenber (Hrsg.): Informatik 2003 – Mit Sicherheit Informatik
- P-37 Arndt Bode, Jörg Desel, Sabine Rathmayer, Martin Wessner (Hrsg.): DeLFI 2003: e-Learning Fachtagung Informatik
- P-38 E.J. Sinz, M. Plaha, P. Neckel (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2003
- P-39 Jens Nedon, Sandra Frings, Oliver Göbel (Hrsg.): IT-Incident Management & IT-Forensics – IMF 2003
- P-40 Michael Rebstock (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2004
- P-41 Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle, Thomas Runkler (Edts.): ARCS 2004 – Organic and Pervasive Computing
- P-42 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Economy – Transaktionen und Prozesse, Anwendungen und Dienste
- P-43 Birgitta König-Ries, Michael Klein, Philipp Obreiter (Hrsg.): Persistence, Scalability, Transactions – Database Mechanisms for Mobile Applications
- P-44 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): Security, E-Learning, E-Services
- P-45 Bernhard Rumpe, Wolfgang Hesse (Hrsg.): Modellierung 2004
- P-46 Ulrich Flegel, Michael Meier (Hrsg.): Detection of Intrusions of Malware & Vulnerability Assessment
- P-47 Alexander Prosser, Robert Krimmer (Hrsg.): Electronic Voting in Europe – Technology, Law, Politics and Society
- P-48 Anatoly Doroshenko, Terry Halpin, Stephen W. Liddle, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications
- P-49 G. Schiefer, P. Wagner, M. Morgenstern, U. Rickert (Hrsg.): Integration und Datensicherheit – Anforderungen, Konflikte und Perspektiven
- P-50 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 1) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm
- P-51 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 2) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm
- P-52 Gregor Engels, Silke Seehusen (Hrsg.): DELFI 2004 – Tagungsband der 2. e-Learning Fachtagung Informatik
- P-53 Robert Giegerich, Jens Stoye (Hrsg.): German Conference on Bioinformatics – GCB 2004
- P-54 Jens Borchers, Ralf Kneuper (Hrsg.): Softwaremanagement 2004 – Outsourcing und Integration
- P-55 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): E-Science und Grid Ad-hoc-Netze Medienintegration
- P-56 Fernand Feltz, Andreas Oberweis, Benoit Otjacques (Hrsg.): EMISA 2004 – Informationssysteme im E-Business und E-Government
- P-57 Klaus Turowski (Hrsg.): Architekturen, Komponenten, Anwendungen
- P-58 Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner, Franz Schweiggert (Hrsg.): Testing of Component-Based Systems and Software Quality
- P-59 J. Felix Hampe, Franz Lehner, Key Pousttchi, Kai Rannenber, Klaus Turowski (Hrsg.): Mobile Business – Processes, Platforms, Payments
- P-60 Steffen Friedrich (Hrsg.): Unterrichtskonzepte für informatische Bildung
- P-61 Paul Müller, Reinhard Gotzhein, Jens B. Schmitt (Hrsg.): Kommunikation in verteilten Systemen
- P-62 Federrath, Hannes (Hrsg.): „Sicherheit 2005“ – Sicherheit – Schutz und Zuverlässigkeit
- P-63 Roland Kaschek, Heinrich C. Mayr, Stephen Liddle (Hrsg.): Information Systems – Technology and its Applications

- P-64 Peter Liggesmeyer, Klaus Pohl, Michael Goedicke (Hrsg.): Software Engineering 2005
- P-65 Gottfried Vossen, Frank Leymann, Peter Lockemann, Wolffried Stucky (Hrsg.): Datenbanksysteme in Business, Technologie und Web
- P-66 Jörg M. Haake, Ulrike Lucke, Djamshid Tavangarian (Hrsg.): DeLFI 2005: 3. deutsche e-Learning Fachtagung Informatik
- P-67 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 1)
- P-68 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 2)
- P-69 Robert Hirschfeld, Ryszard Kowalczyk, Andreas Polze, Matthias Weske (Hrsg.): NODe 2005, GSEM 2005
- P-70 Klaus Turowski, Johannes-Maria Zaha (Hrsg.): Component-oriented Enterprise Application (COAE 2005)
- P-71 Andrew Torda, Stefan Kurz, Matthias Rarey (Hrsg.): German Conference on Bioinformatics 2005
- P-72 Klaus P. Jantke, Klaus-Peter Fährnrich, Wolfgang S. Wittig (Hrsg.): Marktplatz Internet: Von e-Learning bis e-Payment
- P-73 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): "Heute schon das Morgen sehen"
- P-74 Christopher Wolf, Stefan Lucks, Po-Wah Yau (Hrsg.): WEWoRC 2005 – Western European Workshop on Research in Cryptology
- P-75 Jörg Desel, Ulrich Frank (Hrsg.): Enterprise Modelling and Information Systems Architecture
- P-76 Thomas Kirste, Birgitta König-Riess, Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Informationssysteme – Potentiale, Hindernisse, Einsatz
- P-77 Jana Dittmann (Hrsg.): SICHERHEIT 2006
- P-78 K.-O. Wenkel, P. Wagner, M. Morgens-tern, K. Luzi, P. Eisermann (Hrsg.): Land- und Ernährungswirtschaft im Wandel
- P-79 Bettina Biel, Matthias Book, Volker Gruhn (Hrsg.): Softwareengineering 2006
- P-80 Mareike Schoop, Christian Huemer, Michael Rebstock, Martin Bichler (Hrsg.): Service-Oriented Electronic Commerce
- P-81 Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle (Hrsg.): ARCS '06
- P-82 Heinrich C. Mayr, Ruth Breu (Hrsg.): Modellierung 2006
- P-83 Daniel Huson, Oliver Kohlbacher, Andrei Lupas, Kay Nieselt and Andreas Zell (eds.): German Conference on Bioinformatics 2005
- P-84 Dimitris Karagiannis, Heinrich C. Mayr, (Hrsg.): Information Systems Technology and its Applications
- P-85 Witold Abramowicz, Heinrich C. Mayr, (Hrsg.): Business Information Systems
- P-86 Robert Krimmer (Ed.): Electronic Voting 2006
- P-87 Max Mühlhäuser, Guido Röbling, Ralf Steinmetz (Hrsg.): DELFI 2006: 4. e-Learning Fachtagung Informatik
- P-88 Robert Hirschfeld, Andreas Polze, Ryszard Kowalczyk (Hrsg.): NODe 2006, GSEM 2006
- P-90 Joachim Schelp, Robert Winter, Ulrich Frank, Bodo Rieger, Klaus Turowski (Hrsg.): Integration, Informationslogistik und Architektur
- P-91 Henrik Stormer, Andreas Meier, Michael Schumacher (Eds.): European Conference on eHealth 2006
- P-92 Fernand Feltz, Benoît Otjacques, Andreas Oberweis, Nicolas Poussing (Eds.): AIM 2006
- P-93 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 1
- P-94 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 2
- P-95 Matthias Weske, Markus Nüttgens (Eds.): EMISA 2005: Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen
- P-96 Saartje Brockmans, Jürgen Jung, York Sure (Eds.): Meta-Modelling and Ontologies
- P-97 Oliver Göbel, Dirk Schadt, Sandra Frings, Hardo Hase, Detlef Günther, Jens Nedon (Eds.): IT-Incident Mangament & IT-Forensics – IMF 2006

- P-98 Hans Brandt-Pook, Werner Simonsmeier und Thorsten Spitta (Hrsg.): Beratung in der Softwareentwicklung – Modelle, Methoden, Best Practices
- P-99 Andreas Schwill, Carsten Schulte, Marco Thomas (Hrsg.): Didaktik der Informatik
- P-100 Peter Forbrig, Günter Siegel, Markus Schneider (Hrsg.): HDI 2006: Hochschuldidaktik der Informatik
- P-101 Stefan Böttinger, Ludwig Theuvsen, Susanne Rank, Marlies Morgenstern (Hrsg.): Agrarinformatik im Spannungsfeld zwischen Regionalisierung und globalen Wertschöpfungsketten
- P-102 Otto Spaniol (Eds.): Mobile Services and Personalized Environments
- P-103 Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, Christoph Brochhaus (Hrsg.): Datenbanksysteme in Business, Technologie und Web (BTW 2007)
- P-104 Birgitta König-Ries, Franz Lehner, Rainer Malaka, Can Türker (Hrsg.) MMS 2007: Mobilität und mobile Informationssysteme
- P-105 Wolf-Gideon Bleek, Jörg Raasch, Heinz Züllighoven (Hrsg.) Software Engineering 2007
- P-106 Wolf-Gideon Bleek, Henning Schwentner, Heinz Züllighoven (Hrsg.) Software Engineering 2007 – Beiträge zu den Workshops
- P-107 Heinrich C. Mayr, Dimitris Karagiannis (eds.) Information Systems Technology and its Applications
- P-108 Arslan Brömme, Christoph Busch, Detlef Hühnlein (eds.) BIOSIG 2007: Biometrics and Electronic Signatures
- P-109 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 1
- P-110 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 2
- P-111 Christian Eibl, Johannes Magenheimer, Sigrid Schubert, Martin Wessner (Hrsg.) DeLFI 2007: 5. e-Learning Fachtagung Informatik
- P-112 Sigrid Schubert (Hrsg.) Didaktik der Informatik in Theorie und Praxis
- P-113 Sören Auer, Christian Bizer, Claudia Müller, Anna V. Zhdanova (Eds.) The Social Semantic Web 2007 Proceedings of the 1st Conference on Social Semantic Web (CSSW)
- P-114 Sandra Frings, Oliver Göbel, Detlef Günther, Hardo G. Hase, Jens Nedon, Dirk Schadt, Arslan Brömme (Eds.) IMF2007 IT-incident management & IT-forensics Proceedings of the 3rd International Conference on IT-Incident Management & IT-Forensics
- P-115 Claudia Falter, Alexander Schliep, Joachim Selbig, Martin Vingron and Dirk Walther (Eds.) German conference on bioinformatics GCB 2007
- P-116 Witold Abramowicz, Leszek Maciszek (Eds.) Business Process and Services Computing 1st International Working Conference on Business Process and Services Computing BPSC 2007
- P-117 Ryszard Kowalczyk (Ed.) Grid service engineering and management The 4th International Conference on Grid Service Engineering and Management GSEM 2007
- P-118 Andreas Hein, Wilfried Thoben, Hans-Jürgen Appelrath, Peter Jensch (Eds.) European Conference on ehealth 2007
- P-119 Manfred Reichert, Stefan Strecker, Klaus Turowski (Eds.) Enterprise Modelling and Information Systems Architectures Concepts and Applications
- P-120 Adam Pawlak, Kurt Sandkuhl, Wojciech Cholewa, Leandro Soares Indrusiak (Eds.) Coordination of Collaborative Engineering - State of the Art and Future Challenges
- P-121 Korbinian Herrmann, Bernd Bruegge (Hrsg.) Software Engineering 2008 Fachtagung des GI-Fachbereichs Softwaretechnik
- P-122 Walid Maalej, Bernd Bruegge (Hrsg.) Software Engineering 2008 - Workshopband Fachtagung des GI-Fachbereichs Softwaretechnik

- P-123 Michael H. Breitner, Martin Breunig, Elgar Fleisch, Ley Pousttchi, Klaus Turowski (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Technologien, Prozesse, Marktfähigkeit
Proceedings zur 3. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2008)
- P-124 Wolfgang E. Nagel, Rolf Hoffmann, Andreas Koch (Eds.)
9th Workshop on Parallel Systems and Algorithms (PASA)
Workshop of the GI/ITG Special Interest Groups PARS and PARVA
- P-125 Rolf A.E. Müller, Hans-H. Sundermeier, Ludwig Theuvsen, Stephanie Schütze, Marlies Morgenstern (Hrsg.)
Unternehmens-IT:
Führungsinstrument oder Verwaltungsbürde
Referate der 28. GIL Jahrestagung
- P-126 Rainer Gimnich, Uwe Kaiser, Jochen Quante, Andreas Winter (Hrsg.)
10th Workshop Software Reengineering (WSR 2008)
- P-127 Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.)
Modellierung 2008
- P-128 Ammar Alkassar, Jörg Siekmann (Hrsg.)
Sicherheit 2008
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
2.-4. April 2008
Saarbrücken, Germany
- P-129 Wolfgang Hesse, Andreas Oberweis (Eds.)
Sigsand-Europe 2008
Proceedings of the Third AIS SIGSAND European Symposium on Analysis, Design, Use and Societal Impact of Information Systems
- P-130 Paul Müller, Bernhard Neumair, Gabi Dreö Rodosek (Hrsg.)
1. DFN-Forum Kommunikationstechnologien Beiträge der Fachtagung
- P-131 Robert Krimmer, Rüdiger Grimm (Eds.)
3rd International Conference on Electronic Voting 2008
Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting.CC
- P-132 Silke Seehusen, Ulrike Lucke, Stefan Fischer (Hrsg.)
DeLFI 2008:
Die 6. e-Learning Fachtagung Informatik
- P-133 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik Band 1
- P-134 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik Band 2
- P-135 Torsten Brinda, Michael Fothe, Peter Hubwieser, Kirsten Schlüter (Hrsg.)
Didaktik der Informatik – Aktuelle Forschungsergebnisse
- P-136 Andreas Beyer, Michael Schroeder (Eds.)
German Conference on Bioinformatics GCB 2008
- P-137 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2008: Biometrics and Electronic Signatures
- P-138 Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (Hrsg.)
Synergien durch Integration und Informationslogistik
Proceedings zur DW2008
- P-139 Georg Herzwurm, Martin Mikusz (Hrsg.)
Industrialisierung des Software-Managements
Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschaftsinformatik
- P-140 Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, Dirk Schadt (Eds.)
IMF 2008 - IT Incident Management & IT Forensics
- P-141 Peter Loos, Markus Nüttgens, Klaus Turowski, Dirk Werth (Hrsg.)
Modellierung betrieblicher Informationssysteme (MobIS 2008)
Modellierung zwischen SOA und Compliance Management
- P-142 R. Bill, P. Korduan, L. Theuvsen, M. Morgenstern (Hrsg.)
Anforderungen an die Agrarinformatik durch Globalisierung und Klimaveränderung
- P-143 Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)
Software Engineering 2009
Fachtagung des GI-Fachbereichs Softwaretechnik

- P-144 Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, Gottfried Vossen (Hrsg.)
Datenbanksysteme in Business, Technologie und Web (BTW)
- P-145 Knut Hinkelmann, Holger Wache (Eds.)
WM2009: 5th Conference on Professional Knowledge Management
- P-146 Markus Bick, Martin Breunig, Hagen Höpfner (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Entwicklung, Implementierung und Anwendung
4. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2009)
- P-147 Witold Abramowicz, Leszek Maciaszek, Ryszard Kowalczyk, Andreas Speck (Eds.)
Business Process, Services Computing and Intelligent Service Management
BPSC 2009 · ISM 2009 · YRW-MBP 2009
- P-148 Christian Erfurth, Gerald Eichler, Volkmarr Schau (Eds.)
9th International Conference on Innovative Internet Community Systems
I²CS 2009
- P-149 Paul Müller, Bernhard Neumair, Gabi Dreö Rodosek (Hrsg.)
2. DFN-Forum
Kommunikationstechnologien
Beiträge der Fachtagung
- P-150 Jürgen Münch, Peter Liggesmeyer (Hrsg.)
Software Engineering
2009 - Workshopband
- P-151 Armin Heinzl, Peter Dadam, Stefan Kirn, Peter Lockemann (Eds.)
PRIMIUM
Process Innovation for Enterprise Software
- P-152 Jan Mendling, Stefanie Rinderle-Ma, Werner Esswein (Eds.)
Enterprise Modelling and Information Systems Architectures
Proceedings of the 3rd Int'l Workshop EMISA 2009
- P-153 Andreas Schwill, Nicolas Apostolopoulos (Hrsg.)
Lernen im Digitalen Zeitalter
DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik
- P-154 Stefan Fischer, Erik Maehle, Rüdiger Reischuk (Hrsg.)
INFORMATIK 2009
Im Focus das Leben
- P-155 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2009:
Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures
- P-156 Bernhard Koerber (Hrsg.)
Zukunft braucht Herkunft
25 Jahre »INFOS – Informatik und Schule«
- P-157 Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, Peter Stadler (Eds.)
German Conference on Bioinformatics
2009
- P-158 W. Claupein, L. Theuvsen, A. Kämpf, M. Morgenstern (Hrsg.)
Precision Agriculture
Reloaded – Informationsgestützte Landwirtschaft
- P-159 Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)
Software Engineering 2010
- P-160 Gregor Engels, Markus Luckey, Alexander Pretschner, Ralf Reussner (Hrsg.)
Software Engineering 2010 – Workshopband
(inkl. Doktorandensymposium)
- P-161 Gregor Engels, Dimitris Karagiannis, Heinrich C. Mayr (Hrsg.)
Modellierung 2010
- P-162 Maria A. Wimmer, Uwe Brinkhoff, Siegfried Kaiser, Dagmar Lück-Schneider, Erich Schweighofer, Andreas Wiebe (Hrsg.)
Vernetzte IT für einen effektiven Staat
Gemeinsame Fachtagung
Verwaltungsinformatik (FTVI) und
Fachtagung Rechtsinformatik (FTRI) 2010
- P-163 Markus Bick, Stefan Eulgem, Elgar Fleisch, J. Felix Hampe, Birgitta König-Ries, Franz Lehner, Key Pousttchi, Kai Rannenberg (Hrsg.)
Mobile und Ubiquitäre Informationssysteme
Technologien, Anwendungen und Dienste zur Unterstützung von mobiler Kollaboration
- P-164 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2010: Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

- P-165 Gerald Eichler, Peter Kropf,
Ulrike Lechner, Phayung Meesad,
Herwig Unger (Eds.)
10th International Conference on
Innovative Internet Community Systems
(I²CS) – Jubilee Edition 2010 –
- P-166 Paul Müller, Bernhard Neumair,
Gabi Dreö Rodosek (Hrsg.)
3. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung
- P-167 Robert Krimmer, Rüdiger Grimm (Eds.)
4th International Conference on
Electronic Voting 2010
co-organized by the Council of Europe,
Gesellschaft für Informatik and
E-Voting.CC
- P-168 Ira Diethelm, Christina Dörge,
Claudia Hildebrandt,
Carsten Schulte (Hrsg.)
Didaktik der Informatik
Möglichkeiten empirischer
Forschungsmethoden und Perspektiven
der Fachdidaktik
- P-169 Michael Kerres, Nadine Ojstersek
Ulrik Schroeder, Ulrich Hoppe (Hrsg.)
DeLFI 2010 - 8. Tagung
der Fachgruppe E-Learning
der Gesellschaft für Informatik e.V.
- P-170 Felix C. Freiling (Hrsg.)
Sicherheit 2010
Sicherheit, Schutz und Zuverlässigkeit
- P-171 Werner Esswein, Klaus Turowski,
Martin Jührisch (Hrsg.)
Modellierung betrieblicher
Informationssysteme (MobIS 2010)
Modellgestütztes Management
- P-172 Stefan Klink, Agnes Koschmider
Marco Mevius, Andreas Oberweis (Hrsg.)
EMISA 2010
Einflussfaktoren auf die Entwicklung
flexibler, integrierter Informationssysteme
Beiträge des Workshops der GI-
Fachgruppe EMISA
(Entwicklungsmethoden für Infor-
mationssysteme und deren Anwendung)
- P-173 Dietmar Schomburg,
Andreas Grote (Eds.)
German Conference on Bioinformatics
2010
- P-174 Arslan Brömme, Torsten Eymann,
Detlef Hühnlein, Heiko Roßnagel,
Paul Schmücker (Hrsg.)
perspeGktive 2010
Workshop „Innovative und sichere
Informationstechnologie für das
Gesundheitswesen von morgen“
- P-175 Klaus-Peter Fährnich,
Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für
die Informatik
Band 1
- P-176 Klaus-Peter Fährnich,
Bogdan Franczyk (Hrsg.)
INFORMATIK 2010
Service Science – Neue Perspektiven für
die Informatik
Band 2
- P-177 Witold Abramowicz, Rainer Alt,
Klaus-Peter Fährnich, Bogdan Franczyk,
Leszek A. Maciaszek (Eds.)
INFORMATIK 2010
Business Process and Service Science –
Proceedings of ISSS and BPSC
- P-178 Wolfram Pietsch, Benedikt Krams (Hrsg.)
Vom Projekt zum Produkt
Fachtagung des GI-Fachausschusses
Management der
Anwendungsentwicklung und -wartung
im Fachbereich Wirtschafts-informatik
(WI-MAW), Aachen, 2010
- P-179 Stefan Gruner, Bernhard Rumpe (Eds.)
FM+AM 2010
Second International Workshop on Formal
Methods and Agile Methods
- P-180 Theo Härder, Wolfgang Lehner,
Bernhard Mitschang, Harald Schöning,
Holger Schwarz (Hrsg.)
Datenbanksysteme für Business,
Technologie und Web (BTW)
14. Fachtagung des GI-Fachbereichs
„Datenbanken und Informationssysteme“
(DBIS)
- P-181 Michael Clasen, Otto Schätzel,
Brigitte Theuvsen (Hrsg.)
Qualität und Effizienz durch
informationsgestützte Landwirtschaft,
Fokus: Moderne Weinwirtschaft

The titles can be purchased at:
Köllen Druck + Verlag GmbH
 Ernst-Robert-Curtius-Str. 14 · D-53117 Bonn
 Fax: +49 (0)228/9898222
 E-Mail: druckverlag@koellen.de

Gesellschaft für Informatik e.V. (GI)

publishes this series in order to make available to a broad public recent findings in informatics (i.e. computer science and information systems), to document conferences that are organized in cooperation with GI and to publish the annual GI Award dissertation.

Broken down into

- seminars
- proceedings
- dissertations
- thematics

current topics are dealt with from the vantage point of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure high quality contributions.

The volumes are published in German or English.

Information: <http://www.gi-ev.de/service/publikationen/lni/>

ISSN 1617-5468

ISBN 978-3-88579-274-1