Bernhard Mitschang, Daniela Nicklas, Frank Leymann,
Harald Schöning, Melanie Herschel, Jens Teubner,
Theo Härder, Oliver Kopp, Matthias Wieland (Hrsg.)

Datenbanksysteme für Business, Technologie und Web (BTW 2017)

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

06. – 10.03.2017
in Stuttgart, Deutschland

Gesellschaft für Informatik e.V. (GI)

Lecture Notes in Informatics (LNI) - Proceedings
Series of the Gesellschaft für Informatik (GI)

Volume P-265

ISBN 978-3-88579-659-6
ISSN 1617-5468

Volume Editors

Bernhard Mitschang
Universität Stuttgart
Institut für Parallele und Verteilte Systeme, Abteilung Anwendersoftware
70569 Stuttgart, Germany
E-Mail: bernhard.mitschang@ipvs.uni-stuttgart.de

Daniela Nicklas
Otto-Friedrich-Universität Bamberg
Lehrstuhl für Informatik, insb. Mobile Software Systeme
96047 Bamberg, Germany
E-Mail: daniela.nicklas@uni-bamberg.de

Frank Leymann
Universität Stuttgart
Institut für Architektur von Anwendungssystemen
70569 Stuttgart, Germany
E-Mail: frank.leymann@informatik.uni-stuttgart.de

Harald Schöning
Software AG
Uhlandstr. 12
64297 Darmstadt, Germany
E-Mail: harald.schoening@softwareag.com

Melanie Herschel
Universität Stuttgart
Institut für Parallele und Verteilte Systeme, Abteilung Anwendersoftware
70569 Stuttgart, Germany
E-Mail: Melanie.Herschel@ipvs.uni-stuttgart.de

Jens Teubner
Technische Universität Dortmund
Fakultät für Informatik, Datenbanken und Informationssysteme
44227 Dortmund, Germany
E-Mail: jens.teubner@cs.tu-dortmund.de

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

Oliver Kopp
Universität Stuttgart
Institut für Parallele und Verteilte Systeme, Abteilung Anwendersoftware
70569 Stuttgart, Germany
E-Mail: oliver.kopp@informatik.uni-stuttgart.de
Matthias Wieland
Universität Stuttgart
Institut für Parallele und Verteilte Systeme, Abteilung Anwendersoftware
70569 Stuttgart, Germany
E-Mail: matthias.wieland@informatik.uni-stuttgart.de

printed by Köllen Druck+Verlag GmbH, Bonn

Vorwort

Die 17. Fachtagung „Datenbanksysteme für Business, Technologie und Web“ (BTW) des Fachbereichs „Datenbanken und Informationssysteme“ (DBIS) der Gesellschaft für Informatik (GI) findet vom 6. bis 10. März 2017 an der Universität Stuttgart statt. Die Informatik an der Universität Stuttgart zeichnet sich aus einerseits durch eine breite Praktische Informatik und andererseits durch den Themenfokus Komplexe Informationssysteme. Dazu zählen insbesondere Methoden, Techniken und Werkzeuge für schnelle, sichere und flexible Informationssysteme sowie für die Erstellung und den Betrieb moderner Anwendungen. Der effiziente und intelligente Umgang mit heterogenen Informationen, etwa aus Datenbanken, Sensoren, wissenschaftlichen Experimenten, der Cloud und dem Web spielt dabei eine wesentliche Rolle. Dieses Themenprofil passt perfekt zu dem der BTW und spiegelt sich in vielen Vortragsthemen der BTW 2017 wider.

Auf der BTW trifft sich nun schon seit mehr als 30 Jahren im zweijährigen Rhythmus die deutschsprachige Datenbankgemeinde, um neue Fragestellungen zu erörtern und aktuelle Forschungsergebnisse zu präsentieren und zu diskutieren. Nicht nur Wissenschaftler, sondern auch Praktiker und Anwender finden sich hier zum Wissens- und Erfahrungsaustausch zusammen.

In guter Tradition umfasst auch die BTW 2017 ein wissenschaftliches Programm, ein Industrieprogramm und ein Demonstrationsprogramm sowie ein Studierendenprogramm, verschiedene Workshops und Tutorien. Erstmals neu mit dabei ist eine sog. Data Science Challenge, bei der die Teilnehmer bzw. Teilnehmergruppen einen eigenen Ansatz zur Cloud-basierten Datenanalyse im Rahmen der gegebenen Aufgabenstellung entwickeln werden. Hierzu müssen zur Verfügung gestellte Datenquellen integriert und ausgewertet werden. Die Teilnehmer haben freie Auswahl der verwendeten Cloud-Plattformen und Analysetechniken. Die Data Science Challenge richtet sich speziell an Doktoranden sowie Studierende.

Die aktuellen Einreichungszahlen sind vergleichbar mit denen früherer BTW-Tagungen und spiegeln das nach wie vor große Interesse der Datenbankgemeinde wider. Für das wissenschaftliche Programm wurden nach einem ausführlichen Begutachtungsprozess in einer eintägigen Sitzung des Programmkomitees in Stuttgart aus 45 Einreichungen 29 ausgewählt, davon 18 Lang- und 11 Kurzbeiträge. Ein Kurzbeitrag wurde nach der Begutachtung zurückgezogen, so dass im Tagungsband und auf der Konferenz insgesamt 28 Beiträge präsentiert werden. Die Themenbreite deckt das gesamte BTW-Themenspektrum ab: von Kerndatenbankthemen über Middleware bis zu Web und Cloud. Im Industrieprogramm gab es 16 Einreichungen. Das erfahrene Programmkomitee konnte 13 davon für den vorliegenden Tagungsband und zur Präsentation auf der Tagung auswählen. Das Spektrum der ausgewählten Beiträge reicht dabei von den

Kernbereichen des Datenmanagements über „Big Data und Data Lakes“ bis hin zu dem sehr nachgefragten Anwendungsbereich des „Internet of Things“. Für das Demoprogramm wurden 10 Beiträge eingereicht, die auch vom Programmkomitee zur Live-Demonstration ausgewählt wurden. Die Bandbreite der Demo-Themen spiegelt die gesamte Bandbreite der BTW-Tagung wider und reicht von Applikationsentwicklung bis zu Datenbankarchitekturthemen und Skalierbarkeitsfragen.

Das Hauptprogramm der Tagung wird eingerahmt von drei eingeladenen Vorträgen zu Themen aus dem Bereich der Kerndatenbanktechnologie und zu dem übergreifenden Thema Digitale Plattform und Transformation. Vortragende sind Wolfram Jost (Software AG), Donald Kossmann (Microsoft) und Guy Lohman (ehem. IBM). Auch im Industrieprogramm gibt es eingeladene Hauptbeiträge zu Themen der skalierbaren Verarbeitung großer Datenmengen. Vortragende sind Namik Hrle (IBM) und Christian Sengstock (SAP AG).

Auch dieses Jahr wird wieder ein BTW-Dissertationspreis des GI-Fachbereichs DBIS vergeben. Aus den eingereichten Dissertationen im Zeitraum Oktober 2014 bis November 2016 wurde die Arbeit „Query Processing and Optimization in Modern Database Systems“ von Viktor Leis (TU München) für den Preis ausgewählt.

Im Vorfeld des Hauptprogramms finden Tutorien, Workshops und ein Studierendenprogramm statt. Die Informationen und Materialien zur BTW 2017 werden über die Web-Seiten der Tagung unter http://btw2017.informatik.uni-stuttgart.de/ zur Verfügung stehen.

Die Organisation der BTW-Tagung nebst allen angeschlossenen Veranstaltungen ist nicht ohne die Unterstützung vieler Partner möglich. Diese sind auf den folgenden Seiten aufgeführt; zu ihnen zählen insbesondere die Sponsoren, das Informatik-Forum Stuttgart (infos e.V.), die Universität Stuttgart und auch die Geschäftsstelle der Gesellschaft für Informatik. Ihnen allen gilt unser besonderer Dank!

Stuttgart im Januar 2017

Bernhard Mitschang, Tagungsleitung/Vorsitzender des Organisationskomitees

Daniela Nicklas, Vorsitzende des wissenschaftlichen Programmkomitees

Frank Leymann, Harald Schöning, Vorsitzende des Industrieprogrammkomitees

Melanie Herschel, Jens Teubner, Vorsitzende des Demoprogrammkomitees

Theo Härder, Vorsitzender des Dissertationspreiskomitees

Oliver Kopp, Matthias Wieland, Tagungsband und Organisationskomitee

Tagungsleitung

Bernhard Mitschang, Universität Stuttgart

Organisationskomitee

Johanna Barzen, Univ. Stuttgart
Michael Behringer, Univ. Stuttgart
Uwe Breitenbücher, Univ. Stuttgart
Melanie Herschel, Univ. Stuttgart
Oliver Kopp, Univ. Stuttgart
Frank Leymann, Univ. Stuttgart
Martin Mähler, IBM
Michael Matthiesen, Univ. Stuttgart
Berhard Mitschang, Univ. Stuttgart
Holger Schwarz, Univ. Stuttgart
Matthias Wieland, Univ. Stuttgart
Lena Wiese, Univ. Göttingen

Wissenschaftliches Programm

Vorsitz: Daniela Nicklas, Universität Bamberg

Wolf-Tilo Balke, TU Braunschweig
Andreas Behrend, Univ. Bonn
Carsten Binnig, DHBW Mannheim
Erik Buchmann, KIT Karlsruhe
Stefan Conrad, Univ. Düsseldorf
Christian Decker, HS Reutlingen
Stefan Dessloch, Univ. Kaiserslautern
Jens Dittrich, Univ. Saarland
Markus Endres, Univ. Augsburg
Rainer Gemulla, Univ. Mannheim
Michael Gertz, Univ. Heidelberg
Marco Grawunder, Univ. Oldenburg
Michael Grossniklaus, Univ. Konstanz
Ralf Hartmut Güting, Fernuniv. Hagen
Theo Härder, Univ. Kaiserslautern
Andreas Henrich, Univ. Bamberg
Alfons Kemper, TU München
Meike Klettke, Univ. Rostock
Harald Kosch, Univ. Passau
Wolfgang Lehner, TU Dresden
Ulf Leser, HU Berlin
Klaus Meyer-Wegener, Univ. Erlangen-Nürnberg
Sebastian Michel, TU Kaiserslautern
Ingo Müller, ETH Zürich
Emmanuel Müller, HPI Potsdam
Felix Naumann, HPI Potsdam
Thomas Neumann, TU München
Daniela Nicklas, Univ. Bamberg
Erhard Rahm, Univ. Leipzig
Stefanie Rinderle-Ma, Univ. Vienna
Norbert Ritter, Univ. Hamburg
Gunter Saake, Univ. Magdeburg
Kai-Uwe Sattler, TU Ilmenau
Ingo Schmitt, BTU Cottbus
Holger Schwarz, Univ. Stuttgart
Bernhard Seeger, Univ. Marburg
Thomas Seidl, LMU München
Gunther Specht, Univ. Innsbruck
Uta Störl, Hochschule Darmstadt
Jens Teubner, TU Dortmund
Andreas Thor, HFT Leipzig
Gottfried Vossen, Univ. Münster
Lena Wiese, Univ. Göttingen

Dissertationspreise

Vorsitz: Theo Härder, TU Kaiserslautern

Andreas Heuer, Univ. Rostock
Erhard Rahm, Univ. Leipzig
Thomas Seidl, LMU München
Gerhard Weikum, MPI Saarbrücken

Industrieprogramm

Vorsitz: Frank Leymann, Univ. Stuttgart und Harald Schöning, Software AG

Klaus Bauer, Trumpf
Johanna Barzen, Univ. Stuttgart
Namik Hrle, IBM
Frank Leymann, Univ. Stuttgart
Berthold Reinwald, IBM Research
Ralph Retter, Bosch SoftTec GmbH
Jochen Rütschlin, Daimler AG
Harald Schöning, Software AG

Demoprogramm

Vorsitz: Melanie Herschel, Univ. Stuttgart und Jens Teubner, TU Dortmund

Rainer Gemulla, Univ. Mannheim
Torsten Grust, Univ. Tübingen
Melanie Herschel, Univ. Stuttgart
Katja Hose, Univ. Aalborg
Stefan Manegold, CWI Amsterdam
Thomas Neumann, TU München
Stefanie Scherzinger, HS Regensburg
Jens Teubner, TU Dortmund

Externe Gutachter

Janina Bleicher, LMU München
Kristof Böhmer, Univ. Wien
David Broneske, Univ. Magdeburg
Ziqiang Diao, Univ. Magdeburg
Philipp Egert, BTU Cottbus-Senftenberg
Steffen Friedrich, Univ. Hamburg
Martin Grimmer, Univ. Leipzig
Tobias Hildebrandt, Univ. Wien
Martin Junghanns, Univ. Leipzig
Georg Kaes, Univ. Wien
Daniyal Kazempour, LMU München
Nikolaus Krismer, Univ. Innsbruck
Jens Lechtenbörger, WWU Münster
Fabian Panse, Univ. Hamburg
Martin Pichl, Univ. Innsbruck
Marcus Pinnecke, Univ. Magdeburg
Florian Richter, LMU München
Doris Silbernagl, Univ. Innsbruck
Stefan Sprenger, HU Berlin
Florian Sterz, Univ. Wien
Eva Zangerle, Univ. Innsbruck
Fabio Valdés, Fernuniv. Hagen
Wolfram Wingerath, Univ. Hamburg
Marcel Zierenberg, BTU Cottbus-Senftenberg

Inhaltsverzeichnis

Eingeladene Vorträge

Scientific Program

Query Processing and Languages

Big Data and NoSQL

Data Integration

Data Analytics

Streaming and Dataflows

Cloud and Benchmarks

Scientific Data and Hardware

Index Structures

Dissertation Award Winner

Industrial Program

Big Data

Data Lake

Trends

Internet of Things

Technologie und Anwendung

Demo Program

Autorenverzeichnis

Eingeladene Vorträge

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 21

The Digital Business Platform

Dr. Wolfram Jost[1]

Abstract

Die Digitale Transformation stellt alles auf den Kopf

Digital Business Platform – Das Fundament einer erfolgreichen digitalen Transformation

Wer denkt, dass die Digitalisierung ein neuer Trend ist, der bald wieder vorbei ist, liegt falsch. Die flächendeckende Digitalisierung ist nicht mehr zu stoppen . . . sie hat bereits ganze Industrien in Ihren Manifesten erschüttert und wird sich eher noch schneller und stärker ausbreiten.

Die Zeit drängt und alle Unternehmen müssen sich der Frage stellen, wie sie sich im Digitalen Zeitalter neu orientiert. Einfach eine schlüsselfertige IT Lösung kaufen ist der falsche Weg. Unternehmen müssen sich in mehreren Disziplinen neu ausrichten. Hierzu gehören:

- Transformation
- Integration
- Data
- Visibility
- Applications
- Scalability
- Connectivity

Dr. Wolfram Jost, CTO, Software AG stellt vor, wie eine Digitale Business Platform Ihre digitale Transformation zum Erfolg macht.

[1] Software AG

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 23

Confidentiality à la Carte with Cipherbase

Donald Kossmann[1]

Abstract

Organizations move data and workloads to the cloud because the cloud is cheaper, more agile, and more secure. Unfortunately, the cloud is not perfect and there are some fundamental tradeoffs that need to be made in the cloud. The Cipherbase project studies the tradeoffs between confidentiality and functionality that arise when state-of-the-art cryptography is combined with databases in the cloud: The more operations that are supported on encrypted data, the more information that can be leaked unintentionally. There has been a great deal of work studying these tradeoffs in the specific context of property-preserving encryption techniques. For instance, deterministic encryption can support equality predicates directly over encrypted data, but it is also vulnerable to inference attacks. This talk discusses the tradeoffs that arise in a more general context when trusted computing platforms such as FPGAs or Intel SGX technology are used to process encrypted data.

Joint work with Arvind Arasu, Ken Eguro, Raghav Kaushik, Ravi Ramamurthy and the SQL Server Security Team.

[1] Microsoft Corporation, donaldk@microsoft.com

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 25

Query Optimization – Are We There Yet?

Guy Lohman[1]

Abstract

After nearly 4 decades and hundreds of scientific papers, relational query optimization can hardly be characterized as anything but a huge scientific and commercial success. The market in 2016 for relational database products was estimated by IDC to be about $40B, out of a total database market of $45.1B. And SQL still dominates database application development and is widely recognized as the most successful declarative language. None of this would have been possible without the success of query optimization, which transforms declarative SQL statements of what data the user needs into an "optimal" ***execution plan***, i.e., a detailed, procedural specification for how that data will be accessed and processed.

So are we "there" yet? Are we done? Are all the big and interesting problems solved? Is query optimization as an area of scientific inquiry dead, relegated to incremental improvements and mere engineering? Why do we continue see so many papers on query optimization?

In this talk, I argue that current research appears to be incremental because we are largely attacking the wrong problems while ignoring much harder and more significant problems. We are solving the problems we know how to solve, not the problems that need solving.

Query optimizers are mathematical models of the performance of alternative plans. Any such model that is based upon invalid assumptions or that is not systematically validated throughout its parametric space is not worth the paper on which it is written, because it will inevitably yield wrong results at unknown points in that space. Current commercial optimizers are still largely dependent upon some simplifying assumptions made by the pioneers of query optimization, assumptions that too often are invalid. Yet these optimizers largely get decent plans most of the time because they luckily aren't near the break points between competing plans. We just don't know how bad it really is, because we debug optimizers by exception — that is, when we get an unexpected plan, or a customer complains — rather than by systematic and thorough validation.

Many of these remaining problems caused by invalid assumptions are contained in the Achilles Heel of query optimization: the underlying and ubiquitous ***cardinality model***, which estimates the number of rows resulting from each operation in the execution plan. Examples include the assumptions that constants in predicates are known at optimization time, that join domains enjoy typical key/foreign-key relationships of inclusion, and especially that predicates on columns are probabilistically independent. Additionally, traditional query

[1] BM Almaden Research Center (Retired), guy_lohman@alumni.pomona.edu

optimization cost models assumed that each query runs in isolation from other queries and focussed almost exclusively on the cost of magnetic disk I/Os, the "800-pound gorilla" of early optimizers. But recent advances in large main memories, flash storage, multi-core processors, and highly parallel in-memory database systems necessitate more accurate modeling of all these aspects (simultaneously!). Add to these the challenges of modeling non-relational operations (i.e., user- defined functions on steroids) and data types such as the arrays, repeating groups, and varying schemas of XML and JSON data types, common in Hadoop and now Spark applications, and you have an extensive research agenda.

Paradoxically, increasing the detail of optimizer models in response to these challenges may actually increase the brittleness of an optimizer! This happens because more detailed models inherently incorporate additional assumptions that may be invalid. Yikes! What is a conscientious query optimizer guru to do? I argue that robust and adaptable query plans are superior to optimal ones, that the goal of query optimization is more to avoid the occasionally really bad plan than to ensure the optimal plan, a process that Bruce Lindsay dubbed "goodizing". Accordingly, optimizers should substitute known facts for models whenever possible, an insight that spawned our idea of a "LEarning Optimizer" (LEO).

I will illustrate these problems, and a few possible solutions, with examples and "war stories".

Scientific Program

Query Processing and Languages

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 31

The Complete Story of Joins (in HyPer)

Thomas Neumann[1] Viktor Leis[2] Alfons Kemper[3]

Abstract: SQL has evolved into an (almost) fully orthogonal query language that allows (arbitrarily deeply) nested subqueries in nearly all parts of the query. In order to avoid recursive evaluation strategies which incur unbearable $O(n^2)$ runtime we need an extended relational algebra to translate such subqueries into non-standard join operators. This paper concentrates on the non-standard join operators beyond the classical textbook inner joins, outer joins and (anti) semi joins. Their implementations in HyPer were covered in previous publications which we refer to. In this paper we cover the new join operators *mark-join* and *single-join* at both levels: At the logical level we show the translation and reordering possibilities in order to effectively optimize the resulting query plans. At the physical level we describe hash-based and block-nested loop implementations of these new joins. Based on our database system HyPer, we describe a blue print for the complete query translation and optimization pipeline. The practical need for the advanced join operators is proven by an analysis of the two well known TPC-H and TPC-DS benchmarks which revealed that all variants are actually used in these query sets.

1 Introduction

Joins are arguably the most important relational operators and come in a number of variants. The FROM clause of any SQL query may contain inner joins, as well as left, right, and full outer joins. In addition, many systems support the (anti) semi join operators in order to be able to express (NOT) EXISTS subqueries as joins.

Besides these specific constructs, SQL has become fully orthogonal, i.e., subqueries can occur almost everywhere in a query, including the SELECT, FROM, and WHERE clauses. A query may thus contain an expression which itself contains a subquery and so on. The easiest way to model this is as mutual recursion, i.e., expressions as well as queries can refer to and evaluate each other. Indeed, this is what some systems, for example, PostgreSQL do. The disadvantage of this simple, non-relational approach is that it makes many important optimizations nearly impossible. In effect, it pre-determines the execution plan of common query patterns to nested-loop-style execution with $O(n^2)$ runtime.

We argue that the well-known inner, outer, and semi joins are not enough. To efficiently support full SQL, two additional join types, which in HyPer we call *single join* and *mark join*, are needed. Both variants are introduced in an early stage of the query optimizer in order to translate certain subquery constructs to relational algebra. As a result, the hard-to-optimize mutual recursion of expressions and subqueries is broken up, i.e., expressions do not refer to subqueries any more. Subqueries are translated into re-orderable joins.

[1] Technische Universität München, neumann@in.tum.de
[2] Technische Universität München, leis@in.tum.de
[3] Technische Universität München, kemper@in.tum.de

Our algebra-based, orthogonal approach

- enables additional join-reordering [MN08] opportunities and
- is the foundation of our unnesting [NK15] technique.

This work, therefore, ties together several strands of prior work and gives the full picture of the join optimization pipeline in HyPer. Note that the algorithms of the traditional types have been described in prior work [AKN13, La13, Le14].

The structure of this paper closely mirrors HyPer's query optimizer. After discussing some preliminaries in Section 2, we show how to translate SQL to our extended relational algebra in Section 3. This translation focuses only on correctness, not performance. How this algebra is optimized is described in the following Section 4. Section 5 focuses on the implementation of the non-standard join algorithms.

2 Preliminaries

Before we introduce the non-standard join operators in the next section, let us repeat the definitions of the well-known [Mo14] join variants first.

First, we have the regular *(inner) join*, which is simply defined as cross product followed by a selection:

$$T_1 \bowtie_p T_2 \quad := \quad \sigma_p(T_1 \times T_2).$$

It computes the combination of all matching entries from relation T_1 and relation T_2 using predicate p. It is used in most SQL queries, but its definition is not sufficient in the presence of correlated subqueries. The subquery has to be evaluated for every tuple of the outer query, therefore we define the *dependent join* as

$$T_1 \mathbin{⋈\!\!\!\blacktriangleright}_p T_2 \quad := \quad \{t_1 \circ t_2 | t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\}.$$

Here, the right hand side is evaluated for every tuple of the left hand side. (By convention, only the right hand side may depend not the left hand side.) We denote the attributes produced by an expression T by $\mathcal{A}(T)$, and free variables occurring in an expression T by $\mathcal{F}(T)$. To evaluate the dependent join, $\mathcal{F}(T_2) \subseteq \mathcal{A}(T_1)$ must hold, i.e., the attributes required by T_2 must be produced by T_1. The dependent join and its transformation rules form the basis for HyPer's unnesting, which was described in a prior BTW paper [NK15].

Furthermore, we have the semi, anti semi, the left outer, and full outer joins:

$$T_1 \ltimes_p T_2 \quad := \quad \{t_1 | t_1 \in T_1 \wedge \exists t_2 \in T_2 : p(t_1 \circ t_2)\}$$

$$T_1 \triangleright_p T_2 \quad := \quad \{t_1 | t_1 \in T_1 \wedge \nexists t_2 \in T_2 : p(t_1 \circ t_2)\}$$

$$T_1 ⟕_p T_2 \quad := \quad (T_1 \bowtie_p T_2) \cup \{t_1 \circ_{a \in \mathcal{A}(T_2)} (a : null) | t_1 \in (T_1 \triangleright_p T_2)\}$$

$$T_1 ⟗_p T_2 \quad := \quad (T_1 ⟕_p T_2) \cup \{t_2 \circ_{a \in \mathcal{A}(T_1)} (a : null) | t_2 \in (T_2 \triangleright_p T_1)\}$$

All of these join variants also have a corresponding dependent join variant, which is analogous to the dependent inner join.

Besides the join operators, there is *group by* as additional important operator:

$$\Gamma_{A;a:f}(e) \quad := \quad \{x \circ (a : f(y)) | x \in \Pi_A(e) \wedge y = \{z | z \in e \wedge \forall a \in A : x.a = z.a\}\}$$

It groups its input e (i.e., a base relation or a relation computed from another algebra expression) by A, and evaluates one (or more comma separated) aggregation function(s) to compute aggregated attributes. If A is empty, just one aggregation tuple is produced—as in SQL with a missing group by-clause.

3 Translating Complex SQL to Extended Relational Algebra

The standard operators from the previous section are well known, but they are not sufficient to translate all constructs of modern SQL. Note that, unsurprisingly, some of the examples are a bit unusual. This is to be expected because simple SQL queries can indeed be translated using the well known operations from Section 2. But these somewhat unusual queries are valid SQL, too, and the database needs efficient means to evaluate them. We will illustrate our approach by showing example queries and their translation into relational algebra. We use the following schema:

- Professors: {[Name, <u>PersID</u>, Sabbatical, . . .]}
- Courses: {[Title, ECTS, Lecturer references Professors, . . .]}
- Assistants: {[Name, Boss references Professors, JobTitle, . . .]}

Before discussing the complex cases we briefly cover the so called *canonical translation*, i.e., the textbook mapping of SQL into relational algebra. It simply takes the cross product of the input relations, applies the WHERE predicate as filter, and outputs the result. For example

```
select Title, Name
from Courses, Professors
where PersID = Lecturer
```

is translated into

$$\Pi_{Title,Name}(\sigma_{PersID=Lecturer}(Courses \times Professors)).$$

Later optimization stages will then translate this *canonical translation* into a more efficient plan, for example combining a selection and a cross product into a join. Unfortunately this simple textbook translation is not sufficient for real-world queries which we will now demonstrate. Along the way, we will introduce additional join operators that are needed to handle the various constructs efficiently. Note that for readability reasons we will not include the final projection in the subsequent examples but concentrate on the real operators.

3.1 Dependent Join

The most obvious limitation of the canonical translation is that it ignores correlated subqueries. That is, it just takes the input relations (or input subqueries) from the *from* clause, and forms a cross product from them. This is not always possible in reality. Consider, for example, the following query:

```
select Name, Total
from Professors, (select sum(ECTS) as Total
                  from Courses
                  where PersId = Lecturer) C
```

Here, the subquery depends upon the outer join[4] and a cross product is not applicable. Instead, the correlated subquery has to be added using a *dependent join* like this:

$$Professors \bowtie (\Gamma_{\emptyset,total:sum(ECTS)}(\sigma_{PersId=Lecturer}(Courses))).$$

Of course the query optimizer will try to get rid of the dependent join as quickly as possible, for example using the techniques from [NK15]. But the initial translation step into relation algebra requires a dependent join—using a regular cross product would be incorrect. Some systems use a nested loop join here, without explicitly designating it as *dependent*, but nested loop joins are highly undesirable for performance reasons. It is usually much better to first introduce dependent joins and then convert them into more efficient regular joins using unnesting techniques.

3.2 Single Join

Correlated subqueries are one class of problems during canonical translation, the second class are scalar subqueries. In SQL, subqueries can be used to compute scalar values, as long as they produce exactly one column and at most one row. Consider for example the following query:

[4] Some DBMSs require extra syntax to indicate this correlation, for example `LATERAL` in PostgreSQL, but others like HyPer accept the query as it is.

```
select PersId, p.Name, (select a.Name
                        from Assistants a
                        where a.Boss = p.PersId
                        and JobTitle = 'personal assistant')
from Professors p
```

This query selects the name of the personal assistant for each professor. Remember that we do not want to fall back to mutual recursion, i.e., we do not want to evaluate the subquery for each professor, as that would lead to $O(n^2)$ runtime. Instead, we want to *join* the Professors relation with the subquery, but we have to take the SQL semantics into account: If the subquery produces a single result we use it as scalar value. If the subquery produces no result the scalar value is NULL. And if there is more than one result we have to report an error.

To express this in relational algebra we introduce a new operator, the *single join*. It behaves nearly identical to an outer join, but reports an error if more than one join partner is found:

$$T_1 ⟕^1_p T_2 := \begin{cases} \text{runtime error}, & \text{if } \exists t_1 \in T_1 : (|\{t_1\} ⟕_p T_2| > 1) \\ T_1 ⟕_p T_2, & \text{otherwise} \end{cases}$$

Using this operator, we can translate the scalar subquery into a join:

$$Professors ⟕^1_{\text{true}} σ_{PersId=Boss \wedge JobTitle='personal\ assistant'}(Assistants)$$

And of course the query optimizer will move the (correlated) predicate into the join operator, resulting in

$$Professors ⟕^1_{PersId=Boss} σ_{JobTitle='personal\ assistant'}(Assistants).$$

There are both performance and correctness arguments for introducing the single join. On the performance side a hash-based implemented of the single join ideally has a runtime of $O(n)$, which is much better than the $O(n^2)$ runtime of the recursive evaluation. And in general it is not possible to use other join implementations, as they would not report an error if more than one join partner is found. There are a few exceptions. If the subquery is known to produce at most one tuple, e.g., when binding the primary key or single-tuple aggregation, $⟕$ can be used instead of $⟕^1$. But these are optimizations that are introduced later by the query optimizer, the initial translation step always translates scalar subqueries into single joins.

3.3 Mark Join

Another class of unusual join constructs are predicate subqueries that arise from *exists*, *not exists*, *unique*, and quantified comparisons. Consider for example the following query:

```
select *
from Professors
where exists (select *
              from Courses
              where Lecturer = PersId)
      or Sabbatical = true
```

It would be tempting to translate the subquery into a semi join, and indeed this is what most systems would do without the disjunction, but this is not possible here. We have to output a professor even if no course exists that he or she lectures, and therefore cannot use a semi join.

Instead, we introduce the *mark join*, which creates a new attribute to mark a tuple as having join partners or not:

$$T_1 \rtimes^{M:m}_{p} T_2 \quad := \quad \{t_1 \circ (m : (\exists t_2 \in T_2 : p(t_1 \circ t_2)))|t_1 \in T_1\}$$

Using the mark join, one can translate the query into a relatively normal join query:

$$\sigma_{(m \vee Sabbatical)}(Professors \rtimes^{M:m}_{PersId=Lecturer} Courses)$$

If the marker is used on only conjunctive predicates the query optimizer can usually translate the mark join into semi or anti semi joins. But this is not possible in general, disjunctions, for example, prevent that. Still, the mark join can be evaluated efficiently, usually in $O(n)$ when using hashing, and introducing it is not a problem for the query optimizer.

Note that the semantics of the mark join becomes significantly more subtle than one might think at a first glance when taking NULL values into account. In the following query

```
select Title, ECTS = any (select ECTS from Courses c2
                          where Lecturer = 123) someEqual
from Courses c1
```

which can be translated directly into the following mark join:

$$Courses\ c_1 \rtimes^{M:someEqual}_{c_1.ECTS=c_2.ECTS} \sigma_{c_2.Lecturer=123} Courses\ c_2$$

The result column *someEqual* can have the values TRUE, FALSE, and NULL (i.e., **unknown**). Accordingly, some care is needed to implement the mark join correctly, as we will discuss in Section 5. But the great benefit is that we now translate arbitrary exists/not exists/unique/quantified-comparison queries into a join construct and eventually, after further optimizations, obtain an efficient evaluation strategy.

3.4 Translating SQL Queries

Putting it all together we can now translate arbitrary SQL queries into relational algebra using the following high-level algorithm:

1. translate the *from* clause, from left to right
   a) for each entry produce an operator tree
   b) if there is no correlation combine with the previous tree using $\times$, otherwise use $\bowtie$
   c) the result is a single operator tree
2. translate the *where* clause (if it exists)
   a) for exists/not exists/unique and quantified subqueries add the subquery on top of the current tree using $\bowtie^{M:m}$. Translate the expression itself with m.
   b) for scalar subqueries, introduce $\bowtie^{1}$ and translate the expression with the (single) result column/row.
   c) all other expressions are scalars, translate them directly
   d) the result is added to the top of the current tree using σ
3. translate the *group-by* clause (if it exists)
   a) translate the grouped expressions just like in the *where* clause
   b) the result is added to the top of the current tree using Γ (group-by)
4. translate the *having* clause (if it exists)
   a) logic is identical to the *where* clause
5. translate the *select* clause
   a) translate the result expressions just like in the *where* clause
   b) the result is added to the top of the current tree using Π
6. translate the *order by* clause (if it exists)
   a) translate the result expressions just like in the *where* clause
   b) the result is added to the top of the current tree using a *sort* operator

This procedure translates an arbitrary SQL query into relational algebra, without having to fall back to mutual recursion between operators and expressions. The result can then be optimized by the query optimizer leveraging efficient join implementations, if applicable.

4 Optimizations

Figure 1 gives a high-level overview of HyPer's optimizer. Translating the SQL abstract syntax tree (AST) to relational algebra is done by the *semantic analysis* component. In this step only inner, outer, (left) mark and single joins are introduced. All other variants appear during later optimization phases in order to improve performance.

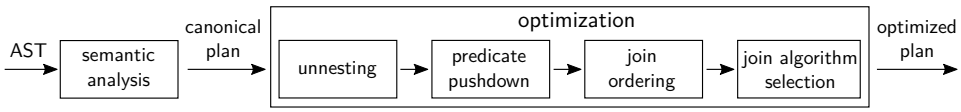


Fig. 1: Overview over optimization process

4.1 Unnesting

One important—and for some queries crucial—optimization is unnesting. For many users, correlated queries are easier to state than semantically equivalent join-based queries. Without unnesting, such queries result in nested-loop joins with $O(n^2)$ runtime. Most systems detect certain correlation patterns and transform them to joins, but are not capable to unnest complex correlations.

Our unnesting technique [NK15], in contrast, is capable of unnesting arbitrary queries—not just some patterns. This is achieved by a set of systematic, algebra-based transformations—as described previously in detail [NK15]. The mark and single join are an implicit building block of our unnesting technique. Without these joins, some queries could not be expressed in relational algebra and, as a result, our unnesting technique would not work.

4.2 Reordering

One of the most important tasks of a query optimizer is to find a good join order since a bad join order can be slower by orders of magnitude [Le15]. As mentioned earlier, in comparison with a mutual recursion-based approach, single and mark joins offer additional reordering opportunities. For example, consider the following query:

```
select *
from Professors p, Assistants a
where p.PersId = a.Boss
      and (exists (select *
                   from Courses c
                   where c.Lecturer = p.PersId)
           or p.Sabbatical = true)
```

Without mark join, the join order is effectively predetermined, and the inner join between *Professors* and *Assistants* is always performed first. Using mark join, on the other hand, it is also possible to start with the mark join before joining with Assistants:

$$(\sigma_{(m \vee Sabbatical)}(Professors \bowtie^{M:m}_{PersId=Lecturer} Courses)) \bowtie_{PersId=Boss} Assistants$$

If there are more Assistants than Professors, performing the mark join first is faster than starting with the inner join. Due to transitivity of the join predicates (*PersId* = *Boss* and *Lecturer* = *PersId*), it is also possible to start with a mark join between Courses and

Assistants. The decision between the three join orders is done by the cost-based join enumeration algorithm. HyPer uses a graph-based dynamic programming algorithm called *DPhyp* that only enumerates connected components without cross products [MN08]. The algorithm takes ordering constraints of non-inner joins into account.

4.3 Left and Right Join Variants

For most join types HyPer has a left (e.g., left mark join) and a right (e.g., right mark join) variant. Both variants semantically produce the same result (with left and right inputs swapped). They differ, however, in their performance.

In hash-based execution, for example, a hash table is built from the left[5] input (the build input). A tuple from the right input will (the probe input) result in a hash table lookup in this hash table. Because hash table insertion is usually slower than lookup, for performance reasons, HyPer's query optimizer swaps the argument order of joins such that the smaller[6] input is on the left. To summarize, having two variants for each join gives the optimizer the freedom that leads to better query performance.

4.4 Other Optimizations

Mark joins are slightly slower than (anti) semi joins because they have to maintain a marker. If possible, mark joins should therefore be translated to (anti) semi joins. In HyPer, the query

```
select *
from Professors
where exists (select *
              from Courses
              where Lecturer = PersId)
```

is first expressed using a mark join, that is then replaced by a semi join ($Professors \ltimes Courses$) in a later optimization step.

Another optimization is to translate outer joins to inner joins. This is possible in the presence of null-rejecting predicates as in the following example:

```
select Title, Name
from Courses right outer join Professors on PersID = Lecturer
where ECTS > 1
```

Finally, a left single join can be replaced by a normal left outer join if the subquery is known to compute at most one row as in the following example:

[5] Note that contrary to our convention, some systems hash the right side.
[6] This is done based on cardinality estimates.

```
select Name, (select sum(ECTS) as Total
              from Courses
              where PersId = Lecturer)
from Professors
```

In this query, a normal outer join is sufficient because the subquery is an aggregate without a group by clause, which always produces a single row.

5 Algorithms

After discussing the various join variants and their optimization, we now come to their actual implementation. Note that we are primarily interested in the high-level algorithm, that is, we do not introduce a particularly tuned implementation, but rather discuss how they differ from regular joins. We therefore describe them in terms of simple main-memory algorithms. The generalization from that to, e.g., external memory algorithms is relatively simple. We will start the discussion with equi-joins, as they are the most common joins and can be implemented efficiently, and then cover non-equi joins.

5.1 Regular Equi-Joins

To highlight the differences between joins, we start with regular hash-based equi joins. Because we describe only the in-memory case here, the code is relatively short, and serves as basis for the different variants. For simplicity we assume that we compute $R \bowtie_{a=b} S$. This can be implemented as follows:

List. 1: Equality Hash Join

```
for each r in R
   store r into H[r.a]
for each s in S
   for each r in H[s.b]
      if r.a = s.b
         emit r,s
```

A real implementation will be much more involved [AKN13, La13, Le14], of course, but this pseudo code helps to illustrate the basic algorithm: A hash table holds all tuples from one side, organized by the join attribute, and the other side probes that hash table to find join partners.

5.2 Joins with Mixed Types

Even the simple equi-join becomes more involved if mixed data types are involved. In our $R \bowtie_{a=b} S$ example, how should we organize the hash table if, e.g., a has the data type

`numeric(6,3)` and *b* has the data type `integer`? The internal representation of numbers will be quite different, but still we have to make sure that 3 joins with 3.000, but not with 3.001. This will usually not be the case when using the native hash functions of the different data types.

The key insight here is that one should perform the join on the *most restrictive* data type, in our example `integer`. Every value that cannot be represented exactly as an integer will never have a join partner, and thus can be omitted from the hash table. In pseudo code this can be expressed like this (assuming *b* has the most restrictive data type):

List. 2: Equality Hash Join with Mixed Types

```
for each r in R
   a' = r.a cast to the type of S.b
   a'' = a' cast to the type of R.a
   if r.a = a''
      store r into H[a']
for each s in S
   for each r in H[s.b]
      if r.a = s.b
         emit r,s
```

Note that while the pseudo code here assumes that *a* is cast to the type of *b*, it could also be the other way round. For the following algorithms we call the most restrictive data type "compare type" and will shorten that logic to "if cast was exact".

5.3 Outer Joins

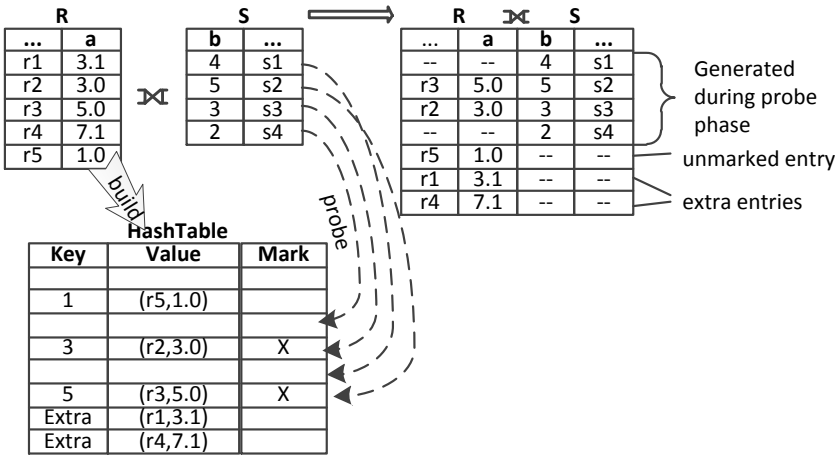


Fig. 2: Outer Join Example

Outer joins output the same tuples as inner joins, and in addition also output all tuples that did not find join partners. This can be implemented by *marking* tuples. As illustrated in Figure 2, each hash table entry has one additional byte, initially 0, that is set to 1 when a join partner is found. This allows us to output all tuples that did not have a join partner after the join is done. For the tuples from the right-hand side we know immediately if we had a join partner or not. We show the pseudo code for the full outer join below, the left and right outer joins are the straightforward subsets of the algorithm.

List. 3: Full Outer Hash Join

```
for each r in R
   a' = r.a cast to compare type
   if cast was exact
      store r into H[a']
   else
      store r into H["extra"]
for each s in S
   sm = 0
   b' = s.b cast to compare type
   if cast was exact
      for each r in H[b']
         if r.a = s.b
            mark r as joined
            sm = 1
            emit r,s
   if sm = 0
      emit null,s
for each r in H
   if r is not marked
      emit r,null
```

The code first processes all R tuples and stores them (unmarked) in the hash table. One extra difficulty here is that some R values potentially cannot be represented in the *compare type* when mixing data types. For inner joins we could drop these, but for left and full outer joins we must produce them anyway, and thus store them in an *extra* hash table bucket that is never selected by the hash function.

Afterwards, we process all S tuples, initialize their local marker (*sm*) to zero, and probe the hash table for potential candidates. When we find one, we mark both the S and the R tuple as having join partners before emitting the joined tuple. After that hash table probe, we know whether the S tuple has a join partner; if not, we emit it after padding it with NULL. After processing the S tuples, we do one sweep over the hash table H and emit all tuples that did not have join partners. This includes the *extra* tuples from data type mismatches.

5.4 (Anti) Semi Joins

This idea of marking tuples can be extended to (anti) semi joins. In semi joins, we output a tuple only if it has not been marked before. In anti semi joins, we emit the tuples afterwards that did not have join partners. These are largely variants of the full outer join code shown above, for space reasons we only show the left semi join:

List. 4: Left Semi Hash Join

```
for each r in R
   a' = r.a cast to compare type
   if cast was exact
      store r into H[a']
for each s in S
   b' = s.b cast to compare type
   if cast was exact
      for each r in H[b']
         if r.a = s.b and r is not marked
            mark r as joined
            emit r
```

The semi join code is not very complex, but some care is needed when handling the marker. If the semi join is executed multi-threaded, only one thread must be allowed to emit a particular r value. The easiest way to achieve this is to update the marker using atomic instructions. Anti semi joins mark the tuple, but do not emit it, and instead add a pass at the end to emit unmarked tuples.

5.5 Single Joins

Single joins use the same marking mechanism as left outer joins, and in addition use the marker to detect multiple output values. This is a very attractive way to implement single joins, because in practice this means that the single join is for free, it costs virtually the same as an outer joins. The left single join variant ($⟕^1$) is shown below, it is basically an extension of the left outer join.

List. 5: Left Single Hash Join

```
for each r in R
   a' = r.a cast to compare type
   if cast was exact
      store r into H[a'] else
      store r into H["extra"]
for each s in S
   b' = s.b cast to compare type
   if cast was exact
```

```
      for each r in H[b']
         if r.a = s.b
            if r was marked as joined
               throw an exception
            mark r as joined
            emit r,s
for each r in H
   if r is not marked
      emit r,null
```

Analogously, the right single join is an extension of the right outer join that throws an exception if more than one join partner is found.

5.6 Mark Joins

While the mark join uses a similar marking mechanism as the other algorithms, it is more complex because the marker may not only be TRUE (had a join partner) or FALSE (had no join partner), but also NULL (had a join partner where the comparison result is NULL, but none where the comparison is TRUE). This complicates the code considerably. In cases where the NULL implicitly behaves like a FALSE (e.g., in a disjunctive WHERE clause) the NULL case can be simplified to be identical to the FALSE case, but in the general case extra logic for NULL is needed. The code for the left mark join is shown below:

List. 6: Left Mark Hash Join

```
for each r in R
   if r.a is NULL
      mark r as NULL
      store r into H["null"]
   else
      mark r as FALSE
      a' = r.a cast to compare type
      if cast was exact
         store r into H[a'] else
         store r into H["extra"]
hadNull = false
for each s in S
   if s.b is NULL
      hadNull = true
   else
      b' = s.b cast to compare type
      if cast was exact
         for each r in H[b']
            if r.a = s.b
               mark r as TRUE
```

```
if |S| = 0
   mark all entries in H["null"] as FALSE
for each r in H
   if r is marked FALSE and hadNull
      mark r as NULL
   emit r,marker of r
```

There are multiple subtleties here: First, we now need two extra lists. One list for the values outside the domain of the comparison type, marked as FALSE (i.e., having no join partner). And one list for the tuples where the join attribute is NULL, as all comparisons with this tuple will result in NULL. We do not perform this comparison, but instead statically mark them as NULL and put them in an extra list.

During the join itself, we check for NULL values in the join attribute of S. If we encounter a NULL value, we know that each output tuple will either have the marker TRUE or NULL, but never FALSE (as the NULL value would "join" with all of them). If it is not null, we do the hash table lookup and mark all matching tuples with TRUE.

Afterwards, we scan the hash table and output all tuples with their respective markers. Again there are two special cases: If we did not see any tuples in S at all, the initial NULL marker from the "null" list must be converted into FALSE. And if a tuple is marked as FALSE, but we did see a NULL value in S, the whole tuple is now marked as NULL, as the NULL value would have implicitly joined with it. The code for the right mark join is analogous, except that we now mark the right hand side.

5.7 Non-Equi Joins

While equi-joins are the most common, they are not the only kind of joins. In general, a join predicate can be an arbitrary expression, which cannot always be evaluated using a hash join. Before coming to the general case, let us first discuss "nearly" equality joins, for example joins of the form $R \bowtie_{a=b \wedge c>d} S$. These predicates have an equality component that can be evaluated using hash joins, and doing so is usually a good idea. However, some care must be taken for the non-equal part. For an inner join it is possible to split the predicate, i.e., $R \bowtie_{a=b \wedge c>d} S \equiv \sigma_{c>d}(R \bowtie_{a=b} S)$, which can be answered easily. But for outer joins, for example, this split is not possible, and the additional restrictions must be evaluated directly during the join to avoid incorrect results. For compiling database systems like HyPer [Ne11] this combined evaluation is quite natural, but systems like Vectorwise [ZB12] require extra logic to evaluate arbitrary expressions during a hash join. Note that this non-equality part of the join condition can also return NULL as result, which is relevant for the mark join and results in a NULL marker if the current marker is FALSE.

For arbitrary predicates hash joins are not an option, and the database system needs algorithms for these, too. The best choice is usually a blockwise nested loop join, where chunks of R are loaded into memory and joined with the tuples from S. Note that while this algorithm has the same asymptotic cost as a naive nested loop join, it is much faster in

practice, often by orders of magnitude. The pseudo code for the full outer join is shown below, the other join types follow analogously:

List. 7: Full Outer Non-Equality Blockwise Nested Loop Join

```
for each r in R
   if B is full
      call joinBuffer(B)
   store r into B
if B is not empty
   call joinBuffer(B)
for each s in S
   if S is not marked
      emit null,s

function joinBuffer(B):
for each s in S
   for each r in B
      if p(r,s)
         mark r as joined
         mark s as joined
         emit r,s
for each r in B
   if r is not marked
      emit r,null
clear B
```

The join initializes an empty buffer, and then loads as many tuples from R into the buffer as possible. When the buffer is full, *joinBuffer* is called, which joins all tuples from S with the current buffer content, marks join partners and emits results. After reading S, all unmarked tuples from the buffer are emitted padded with NULL, and the buffer is cleared. This continues until R has been processed completely. Finally, all unmarked tuples from S are emitted padded with NULL.

While this is a different algorithm, the marking uses the same logic as in the equality case. The main question is, how do we implement these markers? Marking the left side is easy, we just use one byte per tuple in the buffer for marking. But marking the right hand side is difficult, as we read the right hand side multiple times and do not materialize it in memory. One could maintain an extra vector for this and spool it to disk as needed, but this is expensive. HyPer instead uses an associative data structure that assigns a bit value to each tuple and that uses interval compression. The idea is that often either very few tuples qualify or nearly all tuples qualify, which both result in small data structures due to the interval compression.

6 Experiments

Evaluating the approach from this paper in a meaningful way is not easy, as the conversion from an approach with mutual recursion into one with specialized joins often transforms an $O(n^2)$ algorithm into an $O(n)$ algorithm. As such, numerical comparisons are largely pointless, because the differences get huge even for modest data sizes. In the following we therefore first show that these kinds of joins occur in widely used benchmarks, and then emphasize that the runtime effects of avoiding mutual recursion can be arbitrarily large.

We compare two systems: HyPer, which implements the techniques and algorithms described in this paper, and PostgreSQL, which implements the mutual recursion strategy. When comparing the query execution times of the two systems it is, however, hard to disentangle the effects (query engine performance, join types, optimizations) using complex benchmark queries. We therefore show some simple example queries and discuss how these queries are evaluated and how well they perform.

	TPC-H		TPC-DS	
	before opt.	after opt.	before opt.	after opt.
inner $\bowtie$	yes	yes	yes	yes
left outer $\mathbin{⟕}$	yes	no	yes	yes
right outer $\mathbin{⟖}$	no	no	no	yes
full outer $\mathbin{⟗}$	no	no	yes	yes
single $\bowtie^1$	no	no	yes	yes
left mark $\bowtie^M$	yes	no	yes	yes
right mark $\mathbin{⟖}^M$	-	no	-	yes
left semi $\ltimes$	-	yes	-	yes
right semi $\rtimes$	-	yes	-	yes
left anti semi $\triangleright$	-	yes	-	yes
right anti semi $\triangleleft$	-	yes	-	no
group join	-	yes	-	yes

Tab. 1: Join types occurring in TPC-H and TPC-DS before optimization and after optimization. The entries marked as "-" are never introduced by the translation phase (before optimization).

For each join variant, Table 1 shows whether this join occurs in TPC-H and TPC-DS. The table also distinguishes between the plan canonical translation ("before opt.") and the optimized plan ("after opt."). Incidentally, in the TPC queries *all* join types occur, either before or after optimization. There are fewer TPC-H than TPC-DS queries, and they are generally less complex. In TPC-H, single joins do not occur and all the left mark joins can be translated to 4 left/right semi (anti) join variants. In TPC-DS, on the other hand, both single and mark joins are sometimes needed even after optimization. The table also shows that both the right and left variants are chosen by the query optimizer. To summarize, Table 1 indicates that the "zoo" of join variants is indeed needed and a query optimizer will benefit from having all these variants.

Let us now turn to a simple example query on the TPC-H data set (scale factor 1) that illustrates the performance benefit of the single join:

```
select p_name,
       (select l_orderkey
        from lineitem
        where l_partkey = p_partkey
        and l_returnflag = 'R' and l_linestatus = 'O')
from part
```

In HyPer, this query is evaluated in 17 ms (with 1 thread), while PostgreSQL requires 26 hours. The reason for the abysmal performance of PostgreSQL is that it has to perform a full table scan for each tuple of part, which results in quadratic runtime. Note that some of the techniques described in this paper are required in order to evaluate certain queries to avoid mutual recursive with $O(n^2)$ runtime. Therefore, depending on the data set size, our approach can achieve arbitrary speedups.

7 Prior and Related Work

The SQL translation of Microsoft's SQL server was described in [GJ01]. It covers some of the advanced non-standard joins; but unlike our approach it lacks completeness such that not all query templates can be represented in relational algebra. Therefore, SQL server also cannot unnest all query variants as we described in [NK15]. The work by Hölsch et al. [HGS16] uses the nested relational algebra of the NF^2 model for unnesting transformations.

HyPer uses the prior published join ordering [MN08] and simplification [Ne09] for the logical query optimization. In addition to the logical join operators described here, HyPer also exploits synergies between successive operators; in particular it combines joins and grouping using the group join [MN11].

The physical evaluation of HyPer's joins was described in prior publications: Albutiu et. al. [AKN12] developed the massively parallel sort-merge join MPSM. In a later BTW paper [AKN13] this basic join was extended for outer join and semi join variants. Currently, HyPer is mainly based on parallel hash joins whose parallelization was covered in [La13, Le14].

There is a large body of research on optimizing hash joins for the multi-core arena; most notably the works by Jens Teubner's group [Ba15] and by Jens Dittrich's research group [SCD16] which also investigated the effect of different hash functions [RAD15].

8 Conclusions

Based on our HyPer database system we described a blue print for the complete query translation and optimization pipeline for join queries. As SQL has evolved into an (almost) fully orthogonal query language that allows (arbitrarily deeply) nested subqueries in nearly all parts of the query there is a practical need for adavanced join operators to avoid recursive

evaluation with an unbearable $O(n^2)$ runtime. The practical need for the advanced join operators was proven by an analysis of the two well known TPC-H and TPC-DS benchmarks which revealed that all join variants discussed in this paper are actually used in these query sets. The paper covered the logical query translation and optimization as well as the physical algorithmic implementation of these join operators. Here, we concentrated on the hash join and block nested loop join variants which are predominantly used in HyPer's current query engine. For practical "hands-on" evaluation the interested readers are invited to experiment with our on-line demo at `hyper-db.de` that provides a graphical visualization of the step-by-step query translation and optimization phases.

References

[AKN12] Albutiu, Martina-Cezara; Kemper, Alfons; Neumann, Thomas: Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. PVLDB, 5(10):1064–1075, 2012.

[AKN13] Albutiu, Martina-Cezara; Kemper, Alfons; Neumann, Thomas: Extending the MPSM Join. In: Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings. pp. 57–71, 2013.

[Ba15] Balkesen, Cagri; Teubner, Jens; Alonso, Gustavo; Özsu, M. Tamer: Main-Memory Hash Joins on Modern Processor Architectures. IEEE Trans. Knowl. Data Eng., 27(7):1754–1766, 2015.

[GJ01] Galindo-Legaria, César A.; Joshi, Milind: Orthogonal Optimization of Subqueries and Aggregation. In (Mehrotra, Sharad; Sellis, Timos K., eds): Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001. ACM, pp. 571–581, 2001.

[HGS16] Hölsch, Jürgen; Grossniklaus, Michael; Scholl, Marc H.: Optimization of Nested Queries using the NF2 Algebra. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 1765–1780, 2016.

[La13] Lang, Harald; Leis, Viktor; Albutiu, Martina-Cezara; Neumann, Thomas; Kemper, Alfons: Massively Parallel NUMA-aware Hash Joins. In: Proceedings of the 1st International Workshop on In Memory Data Management and Analytics, IMDM 2013, Riva Del Garda, Italy, August 26, 2013. pp. 1–12, 2013.

[Le14] Leis, Viktor; Boncz, Peter A.; Kemper, Alfons; Neumann, Thomas: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In (Dyreson, Curtis E.; Li, Feifei; Özsu, M. Tamer, eds): International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. ACM, pp. 743–754, 2014.

[Le15] Leis, Viktor; Gubichev, Andrey; Mirchev, Atanas; Boncz, Peter A.; Kemper, Alfons; Neumann, Thomas: How Good Are Query Optimizers, Really? PVLDB, 9(3):204–215, 2015.

[MN08] Moerkotte, Guido; Neumann, Thomas: Dynamic programming strikes back. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008. pp. 539–552, 2008.

[MN11] Moerkotte, Guido; Neumann, Thomas: Accelerating Queries with Group-By and Join by Groupjoin. PVLDB, 4(11):843–851, 2011.

[Mo14] Moerkotte, Guido: Building Query Compiler [Draft]. December 8, 2014.

[Ne09] Neumann, Thomas: Query simplification: graceful degradation for join-order optimization. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009. pp. 403–414, 2009.

[Ne11] Neumann, Thomas: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB, 4(9):539–550, 2011.

[NK15] Neumann, Thomas; Kemper, Alfons: Unnesting Arbitrary Queries. In: Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings. pp. 383–402, 2015.

[RAD15] Richter, Stefan; Alvarez, Victor; Dittrich, Jens: A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. PVLDB, 9(3):96–107, 2015.

[SCD16] Schuh, Stefan; Chen, Xiao; Dittrich, Jens: An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 1961–1976, 2016.

[ZB12] Zukowski, Marcin; Boncz, Peter A.: Vectorwise: Beyond Column Stores. IEEE Data Eng. Bull., 35(1):21–27, 2012.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 51

SPARQLytics: Multidimensional Analytics for RDF

Michael Rudolf[1,2] Hannes Voigt[1] Wolfgang Lehner[1]

Abstract: With the rapid growth of open RDF data in recent years, being able to perform multidimensional analytics with it has become more and more important, in particular for the data analyst performing explorative business intelligence tasks. Existing analytic approaches are often not flexible enough to address the needs of data analysts and enthusiasts with iterative exploratory workflows. In this paper we propose SPARQLytics, a tool that exposes the concepts of multidimensional graph analytics by offering standard OLAP cube operations and generating SPARQL queries. Our evaluation shows that SPARQLytics unburdens data analysts from writing many lines of SPARQL code in iterative data explorations and at the same time it does not impose any overhead to query execution. SPARQLytics fits well with interactive computing tools, such as Jupyter, providing data enthusiasts with a familiar work environment.

1 Introduction

Today, enterprise business intelligence does not rely only on well-controlled in-house data anymore but also makes heavily use of external data sources. Applications, such as marketing research and sentiment analysis, are more and more data-driven. Since they investigate aspects outside of the corporate realm, external data is essential. Hence, the abundance of open data in the web that has become freely available over the last decade is of particular interest. It may allow novel insights that go well beyond the limits of traditional reporting and analysis. 5-star open data,[4] such as the Linked Open Data cloud,[5] follows the RDF data model, links to other open datasets, and is provided as SPARQL endpoints so that it can be directly queried over the web. There is a plethora of such interconnected datasets available.

Open RDF data is particularly useful for data enthusiasts [Mo14], who, on the one hand, do not engage in weekly and monthly reporting but investigate and explore beyond the business segment to discover new trends and business opportunities [Ab13, Ab15]. On the other hand, multidimensional analytics is an important part of the data analyst's toolbox. It is well understood and results can be easily communicated to business colleagues who are familiar with multidimensional reporting. As data enthusiasts explore the unknown, they work iteratively [Bl14]. They play with data, try out initial hypotheses, dismiss some, follow others, dig deeper, refine questions, etc. Every answer can spark new questions.

The traditional OLAP ecosystem offers a wide range of frontend tools for multidimensional analytics. These tools typically speak to an MDX backend provided by a data warehouse

[1] TU Dresden, Database Technology Goup, 01062 Dresden, firstname.lastname@tu-dresden.de
[2] SAP SE, Dietmar-Hopp-Allee 16, 69190 Walldorf, michael.rudolf01@sap.com
[4] See http://5stardata.info/
[5] See http://lod-cloud.net/

with pre-loaded data in a schema designed up-front. As this limits the analytical scenarios to what the schema designer or the metadata provider envisions at design-time, it is not a good fit for a data enthusiast. When working in an ad-hoc iterative exploratory manner, every new question quickly requires new dimensions, new facts, new cubes, or even new data sources. Speaking to a schema designer and re-engineering ETL processes for every other iteration is not a solution.

As a remedy, we propose SPARQLytics, providing the following characteristics:

- SPARQLytics introduces a dedicated component between SPARQL endpoints and the user. It exposes the concepts of multidimensional graph analytics explicitly and facilitates their use in an iterative exploratory work with RDF data.
- SPARQLytics modularizes the ad-hoc definition of multidimensional cubes on RDF data and offers standard OLAP cube operators. Building blocks, such as dimensions and measures, can be easily re-used, reducing SPARQL code writing to a minimum.
- SPARQLytics keeps track of all artifacts in a repository and generates the corresponding SPARQL query that computes measures on a specific cube. It also submits the query to a SPARQL endpoint, hiding most of the technical aspects of the endpoint communication from the user.
- SPARQLytics provides a concise DSL that fits very well with interactive computing tools, such as Jupyter.[6] It builds on SPARQL triple patterns, so that the user can leverage existing SPARQL expertise and quickly reach the productivity break-even.

In Sect. 2 of the paper we present and define the SPARQLytics DSL, while Sect. 3 evaluates it based on the business intelligence workload of the Social Network Benchmark proposed by the Linked Data Benchmark Council (LDBC) [Bo13]. We elaborate on related work in Sect. 4 and conclude the paper in Sect. 5.

2 SPARQLytics

For the SPARQLytics DSL we employ a formalization of multidimensional graph analytics [Ru14] based on the more generic property graph model, which we adapted to the RDF data model. For the sake of readability we re-use production rules from the SPARQL grammar (indicated with angle brackets) where possible.

2.1 Cube Definition

In order to perform OLAP operations, the observations (i.e. facts) to be analyzed and the dimensions and measures of interest need to be defined first. In the remainder of this section we illustrate the various commands provided in our language by means of a running example. To that end we use RDF data generated by the LDBC [Bo13] Social Network Benchmark.[7]

[6] See `http://jupyter.org/`
[7] See `http://ldbcouncil.org/benchmarks/snb`

2.1.1 Fact Selection

In an RDF graph a fact can be an attribute of a vertex or the presence of one or several edges—thus, we can generalize a fact to be a subgraph. The selection of subgraphs from a larger graph can be achieved in various ways, but to us the most convenient one seems to be with the help of graph pattern matching, as it is sufficiently abstract and can be used with a graphical specification or with a dedicated textual language.

In order to facilitate the selection of facts in our language, we re-use certain concepts from SPARQL, because pattern matching is a central building block of it: the prologue steers the resolution of names with the help of prefixes for shortening identifiers and a group graph pattern combines basic graph patters, which consist of triple blocks. The following listing shows the syntax for selecting facts.

```
<Prologue>
SELECT FACTS <GroupGraphPattern>
```

In the following example first three namespace prefixes are defined.[8] Then people who are at least 18 years old (or who will turn 18 this year) are selected as facts for subsequent multidimensional analyses.

```
PREFIX rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX snvoc:   <http://www.ldbc.eu/ldbc_socialnet/1.0/vocabulary/>
PREFIX dbpedia: <http://dbpedia.org/resource/>
SELECT FACTS {
 ?person rdf:type snvoc:Person ;
         snvoc:birthday ?birthday .
 FILTER (YEAR(NOW()) - YEAR(?birthday) >= 18)
}
```

2.1.2 Dimension Specification

The most elementary concept in multidimensional analytics is that of a dimension. It is an aspect of the facts and as such a set of values. In order to better understand the aspect of the facts, the dimensional values are often structured into hierarchies, so that groups of facts can be subsumed by navigating through the levels of the hierarchy. A dimension specification consists of an ordered set of levels and a seed pattern:

```
DEFINE DIMENSION <String> FROM (<TriplesBlock>)
WITH ( LEVEL <String> AS <NumericExpression>, ... )
```

A *seed pattern* is a graph pattern (denoted by the production rule `<TriplesBlock>` from the SPARQL grammar) that is matched against the facts and connects them with the level expressions. This works by binding the variables in the level expression to the values of

[8] The `dbpedia` namespace prefix is not used in the example, it is needed later on.

the equally-named variables in the match of the seed pattern—it is an error if the level expression contains variables not present in the seed pattern. When a level expression is then applied to a match of the seed pattern in a fact, it yields the level member as its value. Note that a seed pattern is not actually necessary for connecting dimensions to facts, because level expressions could be applied directly to the facts. We deliberately introduce them for decoupling facts and dimension specifications in order to enable the re-use of the latter in other analytical scenarios. By encoding in the seed pattern only what is absolutely required from the level expressions, a single dimension specification can easily be applied to a multitude of different facts, provided that the seed pattern matches.

Note the use of names for levels and dimensions—they are required for univocally identifying those constructs within OLAP operations. Although the name of the SPARQL grammar production rule <NumericExpression> suggests a numeric result, the type of level expressions is not restricted to that. It can be used to derive any kind of expression, for which an equivalence relation is defined (i.e., equality comparison has to be supported). The following example specifies the two dimensions `Location` and `Birth Date`.

```
DEFINE DIMENSION "Location" FROM (
  ?person  snvoc:isLocatedIn ?city .
  ?city    snvoc:isPartOf ?country .
  ?country snvoc:isPartOf ?continent
) WITH (
  LEVEL "City" AS ?city,
  LEVEL "Country" AS ?country,
  LEVEL "Continent" AS ?continent
)
```

```
DEFINE DIMENSION "Birth Date" FROM (
  ?person snvoc:birthday ?birthday
) WITH (
  LEVEL "Day" AS DAY(?birthday),
  LEVEL "Month" AS MONTH(?birthday),
  LEVEL "Year" AS YEAR(?birthday)
)
```

In practice it can be desirable to support multiple hierarchies within a single dimension, such as time by calendar year and fiscal year or location by political and topographic division. That would require a partial ordering of dimension levels, but as the same can be achieved with multiple dedicated dimensions sharing the same seed pattern, we deliberately simplify our model for the sake of understanding.

2.1.3 Measure Definition

Measures are numerical values derived from sets of facts and constitute the result of a multidimensional analysis. As with dimension specifications, measure definitions also contain a seed pattern that decouples the derivation of a numerical value from the underlying facts. The following listing shows the syntax for defining a measure with the given name, based on a numeric expression over the variable(s) bound in the specified triples block, with the aggregation performed by the function with the given name.

```
DEFINE MEASURE <String> AS <NumericExpression>
WHERE (<TriplesBlock>) WITH <String>
```

The following example illustrates the definition of two measures: the first measure counts the number of languages a person speaks and is aggregated for groups of people by computing the average, whereas the second measure returns only the maximum length of comments for a single person and also for groups of people.

```
DEFINE MEASURE "Avg. No. Languages"
 AS COUNT(?language) WHERE (
  ?person snvoc:speaks ?language
 ) WITH "AVG"
```

```
DEFINE MEASURE "Max. Comment Length"
 AS MAX(?length) WHERE (
  ?comment snvoc:hasCreator ?person ;
           rdf:type snvoc:Comment ;
           snvoc:length ?length
 ) WITH "MAX"
```

2.1.4 Cube Creation

After the facts, dimensions, and measures have been specified individually and before OLAP operations can be executed, they have to be assembled into a cube. Formally, a graph cube $c := (F, D, M)$ is a triple, where $F := \text{match}(G, p)$ is the set of facts matched by applying the fact selection pattern p to the graph G, D is a set of dimensions, and M is a set of measures. The following listing shows the syntax and an example invocation for creating a cube from the previously specified fact pattern and the dimensions and measures referenced by their names.

```
CREATE CUBE <String> FROM <String>, ... WITH <String>, ...

CREATE CUBE "QB" FROM "Location", "Birth Date"
WITH "Avg. No. Languages", "Max. Comment Length"
```

2.2 OLAP Operations

Similar to SQL, SPARQL is stateless: in between two queries no state is maintained. As a consequence, each multidimensional SPARQL query must contain all information related to the involved facts, dimensions, and measures, which makes them complex and error-prone if written by hand. To simplify multidimensional analytics, we therefore introduce the concept of an *analytical session*, which maintains state in between OLAP operations.

To initiate an analytical session, the data cube that should be subject to the analysis has to be selected from the repository. Since the graph cube is merely a dataset description, the SPARQL endpoint to execute the generated queries against must be specified. The optional dataset clause permits the selection of a specific RDF dataset in case multiple are present:

```
USING CUBE <String> OVER <IRIREF> <DatasetClause>
```

Slicing and dicing are two well-known ways of filtering the facts in a cube. For slicing (i.e. cutting a slice off) a cube, only facts for a specific level member are preserved. The

dice operation accepts a more general predicate selecting a range of level members. The following listing shows the syntax for the two operations with the first two parameters being the names of the dimension and level, respectively.

```
SLICE(<String>, <String>, <PrimaryExpression>)
DICE(<String>, <String> AS <Var>, <ConditionalOrExpression>)
```

An analytical session associates a granularity with the cube, initialized to the lowest level of each dimension. The roll-up operation decreases the granularity associated with the graph cube for the specified dimension whereas the drill-down operation increases it:

```
ROLLUP(<String>, <Integer>)
DRILLDOWN(<String>, <Integer>)
```

Note that similar to the slice/dice operations, the grouping of facts is only modified conceptually by the roll-up and drill-down operations, as the cube's granularity and stored filters are only evaluated when actually computing values for measures.

2.3 Query Generation

To actually have values computed for the measures of interest with the analytical session's current granularity and filters taken into consideration, the user has to issue a corresponding statement listing the names of the desired measures (as in SQL, a star can be specified for all measures). That statement can optionally be accompanied by solution modifiers imposing a sort order on dimensions or measures as well as a limit and an offset, which are helpful for pagination or top-k queries:

```
COMPUTE(* | <String>, ...) ORDER BY <String> ASC | DESC ...
LIMIT <Integer> OFFSET <Integer>
```

This will cause the SPARQL query implementing the fact selection, grouping and aggregation operations and projecting to the given measures to be generated. Note that there are no round trips to the underlying triple store or SPARQL endpoint required for looking up metadata, as all necessary information has been specified before. A typical workflow consists of several measure computations interleaved with changes to the granularity of the cube and its contained facts.

In the following example the first 10 values for the two previously defined measures are computed, ordered by the `Birth Date` dimension:

```
COMPUTE("Avg. No. Languages", "Max. Comment Length")
ORDER BY "Birth Date" ASC LIMIT 10
```

A measure value by itself is meaningless, it needs to be seen in context—the facts that were used in the computation. The group of facts is represented by level members: for each dimension of the graph cube, the corresponding member is returned by applying the expression of the level indicated by the cube's granularity to the fact.

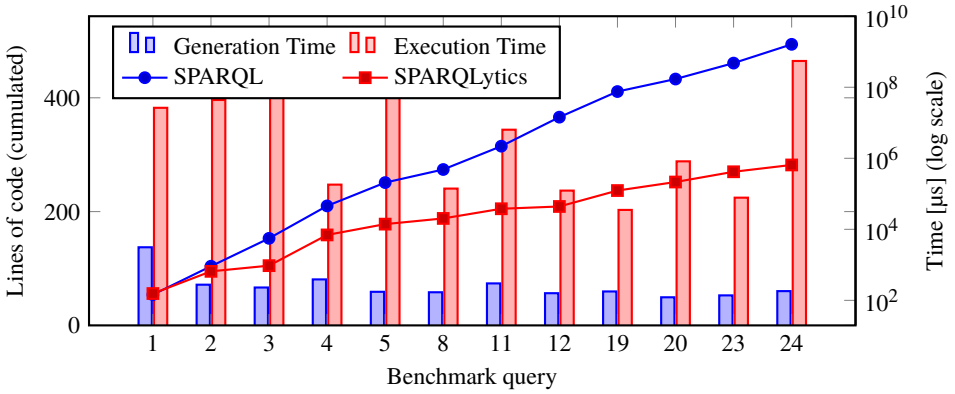


Fig. 1: Accumulated lines of SPARQL and SPARQLytics code for the sequential execution of the implemented benchmark queries (left y-axis) as well as generation and execution times (right y-axis).

3 Evaluation

We have implemented the SPARQLytics DSL described in the previous section in a Java-based prototype using Apache Jena[9] and made it freely available online.[10] A parser translates the commands entered by the user into operations executed in the context of an analytical session, which is backed by a repository of multidimensional artifacts. Whenever the calculation of measures is requested, the corresponding SPARQL query is generated and executed against the chosen endpoint and the results are returned.

As explained in the introduction, the challenge in the big data era lies in use cases that have the human in the loop, and to the best of our knowledge, no benchmark covering such workloads exists today. The Social Network Benchmark proposed by the LDBC [Bo13], is still under development, and at the time of this writing the most recently published version consists only of an interactive workload. However, this benchmark is currently being extended with query descriptions that are typical for business intelligence on graph-structured data.[11] Whereas half of the 24 queries currently proposed express classical data mining tasks, the other half qualify as traditionally multidimensional, identifying dimensions and computing measures. We have implemented those using the SPARQLytics DSL and to that end identified 10 graph cubes, 23 dimensions, and 9 measures. There are at most 8 (on average 3.5) dimensions per cube with 3 (1.5) levels per dimension and 4 (1.6) measures per cube. Dimension and measure artifacts are re-used in up to 3 (on average 1.6) and 5 (1.8) queries, respectively.

Fig. 1 contrasts the cumulated lines of code for SPARQL and SPARQLytics when executing these 12 queries one after another. The plain SPARQL queries are actually not that large—assuming a line width of 120 characters, they average at about 23 lines. However, if a user were to write SPARQL queries for multidimensional analyses by hand, line breaks and indentation would be used to improve readability. Therefore, Fig. 1 shows the number

[9] See `http://jena.apache.org/`.
[10] See `https://github.com/javaprog/sparqlytics`.
[11] See `https://github.com/ldbc/ldbc_snb_implementations`.

of lines of the generated SPARQL queries after having them formatted by Apache Jena. Comparing that to the few lines of code that are needed for executing OLAP operations, such as changing the cube's granularity by rolling up or drilling down, and then invoking the measure computation, it is easy to see the benefit of the abstractions for multidimensional analytics, regardless of whether in a programming interface or on a language level.

The focus of our approach is the generation of SPARQL queries from user-specified RDF cubes modified by OLAP operations, rather than their execution in a SPARQL engine. Therefore we evaluated the effectiveness and efficiency of the generation. Moreover, to verify their correctness, the generated queries were executed on an LDBC dataset with scale factor 1 (3 million entities, 21 million relations) using the Open Source edition of OpenLink Virtuoso[12] in version 7.2.1. We conducted our experiments running both our prototype and the Virtuoso server on a 64 bit Windows 7 workstation with 12 hardware threads and 48 GB RAM. On the right y-axis of Fig. 1 we contrast the query generation and execution times depicted with bars on a logarithmic scale. We assume that the query execution times can be reduced by one order of magnitude with the help of expert performance tuning of Virtuoso. But even then, they would strongly dominate query generation times. Thus, the overhead SPARQLytics imposes on the execution of analytical queries on RDF data is negligible.

Note that the 12-query workload we consider here is just a set of loosely related queries on the same dataset. A real workload induced by a data analyst performing an interactive data exploration is likely to consist of a higher number of queries, which will be considerably more related with a significantly higher re-use of multidimensional artifacts. Every re-use of an artifact directly translates into higher work productivity of the data enthusiast. Also, a stateful interface for interactive multidimensional graph analytics significantly reduces verbosity (i.e. requires less information transfer), because the user has to specify only those aspects of the session's state that should be changed. Avoiding the unnecessary and error-prone repetition leads to a more fluent interaction with the system.

4 Related Work

Several approaches allow feeding RDF graph data often by means of ETL processes into a data warehouse [IAK13, JFL14, KH11, NL12]. A different strategy pursued by more recent approaches is to annotate the RDF data with additional metadata to mark the multidimensional schema within the RDF data or its schema. This metadata enables interpreting the RDF data in a multidimensional model and generating SPARQL queries accordingly [EV12, KOH12, MCG13, Ib14]. All these approaches require the data to be annotated by the provider, who has to fix its intension (i.e., the schema and thereby its meaning) up-front. This works well for homogeneous datasets, such as census data published by government or publicly-owned agencies. Only few organisations do that today, but they limit the available fact bases, dimensions, and measures to what they see fit. As a result, users are not able to turn a measure into a dimension or analyze facts not designated as such, nor can they easily combine different datasets with overlapping semantics.

[12] See `http://virtuoso.openlinksw.com/`.

Recently Colazzo et al. [Co14] proposed to abstract an RDF graph into an analytical schema, which is itself an RDF graph. Each vertex in that schema graph represents a class of facts and has a unary pattern (i.e. one rooted in a single variable) associated to it, whereas every edge stands for instances of concepts captured by binary patterns. Employing the global-as-view metaphor of information integration, the analytical schema can be understood as a "lens" through which the underlying data can be seen. Every vertex then represents a fact base through the pattern attached to it and the reachable vertices represent the available measures and dimensions. This approach is well-suited for exploratory and impromptu analysis: the analytical schema for an RDF dataset has to be constructed by a user and maintained to accommodate for the dynamic nature of graphs, but can also be tailored to a specific analysis task and only extended on demand. A multidimensional query is formulated against the analytical schema but is then automatically rewritten using the attached patterns and executed on the original RDF data. In contrast to our approach, this intentionally does not make multidimensional concepts explicit: the definition of a cube requires writing two queries starting from the same vertex of the schema graph (representing the fact base) and navigating along edges to other vertices (representing the dimensions and measures, respectively). Also, since facts are encoded as unary patterns, they are limited to vertices instead of arbitrary subgraphs (e.g., paths). Nevertheless, the schema graph could be used as a visual guide in graphical user interfaces for exploratory computing scenarios.

5 Conclusion

Using open data sources from the internet for analytical applications is gaining importance for discovering trends and business opportunities outside of the corporate realm—a task typically performed by data enthusiasts. Open RDF data is particularly useful as it offers a plethora of interconnected datasets on nearly every aspect of human life. The traditional approaches for multidimensional analytics on graph-structured data fix the available analytical perspectives up-front and therefore are unsuited for iterative data exploration.

With SPARQLytics we have proposed a tool that exposes the concepts of multidimensional graph analytics for their utilization on RDF data. By modularizing the ad-hoc definition of multidimensional cubes, SPARQLytics makes it simple to define cubes and re-using multidimensional artifacts in modified cube definitions. With standard OLAP cube operators, SPARQLytics allows data analysts to work with the RDF cubes—manipulate them and computing measures on them—with minimal need for SPARQL code writing. For measure computation, SPARQLytics generates the corresponding SPARQL query and posts it to an endpoint, hiding most of the technical aspects of the endpoint communication from the user. SPARQLytics provides a concise DSL that fits very well with interactive computing tools, such as Jupyter, which allows data enthusiasts to leverage SPARQLytics in a familiar environment. The DSL builds on SPARQL triple pattern making it very easy to learn for anyone familiar with RDF, SPARQL, and the concept of multidimensional analytics. Our evaluation shows that SPARQLytics saves data analysts many lines of SPARQL code in iterative data explorations. At the same time, SPARQLytics's query generation in less than 5 ms imposes no overhead to query execution and is practically not recognizable by humans.

References

[Ab13] Abelló, Alberto; Darmont, Jérôme; Etcheverry, Lorena; Golfarelli, Matteo; Mazón, Jose-Norberto; Naumann, Felix; Pedersen, Torben; Rizzi, Stefano Bach; Trujillo, Juan; Vassiliadis, Panos; Vossen, Gottfried: Fusion Cubes: Towards Self-Service Business Intelligence. Int. J. Data Warehous. Min., 9(2):66–88, 2013.

[Ab15] Abelló, Alberto; Romero, Oscar; Bach Pedersen, Torben; Berlanga, Rafael; Nebot, Victoria; Aramburu, María José; Simitsis, Alkis: Using Semantic Web Technologies for Exploratory OLAP: A Survey. IEEE Trans. Knowl. Data Eng., 27(2):571–588, 2015.

[Bl14] Blas, Nicoletta Di; Mazuran, Mirjana; Paolini, Paolo; Quintarelli, Elisa; Tanca, Letizia: Exploratory computing: a draft Manifesto. In: Proc. DSAA. IEEE, pp. 577–580, 2014.

[Bo13] Boncz, Peter: LDBC: Benchmarks for Graph and RDF Data Management. In: Proc. IDEAS. ACM, pp. 1–2, 2013.

[Co14] Colazzo, Dario; Goasdoué, François; Manolescu, Ioana; Roatiş, Alexandra: RDF Analytics: Lenses over Semantic Graphs. In: Proc. WWW. ACM, pp. 467–478, 2014.

[EV12] Etcheverry, Lorena; Vaisman, Alejandro A.: QB4OLAP: A Vocabulary for OLAP Cubes on the Semantic Web. In: Proc. COLD. volume 905 of CEUR Workshop Proceedings. CEUR-WS.org, 2012.

[IAK13] Inoue, Hiroyuki; Amagasa, Toshiyuki; Kitagawa, Hiroyuki: An ETL Framework for Online Analytical Processing of Linked Open Data. In: Proc. Conf. Web-Age Information Management. Springer, pp. 111–117, 2013.

[Ib14] Ibragimov, Dilshod; Hose, Katja; Pedersen, Torben Bach; Zimanyi, Esteban: Towards Exploratory OLAP over Linked Open Data – A Case Study. In: Business Intelligence for the Real-Time Enterprise. volume 206 of LNBIP. Springer, pp. 1–16, 2014.

[JFL14] Jakawat, Wararat; Favre, Cécile; Loudcher, Sabine: OLAP on Information Networks: A New Framework for Dealing with Bibliographic Data. In: Proc. ADBIS, volume 241 of Advances in Intelligent Systems and Computing, pp. 361–370. Springer, 2014.

[KH11] Kämpgen, Benedikt; Harth, Andreas: Transforming Statistical Linked Data for Use in OLAP Systems. In: Proc. International Conference on Semantic Systems. ACM, pp. 33–40, 2011.

[KOH12] Kämpgen, Benedikt; O'Riain, Sean; Harth, Andreas: Interacting with Statistical Linked Data via OLAP Operations. In: Interacting with Linked Data. volume 913 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 36–49, 2012.

[MCG13] Matei, Adriana; Chao, Kuo-Ming; Godwin, Nick: OLAP for Multidimensional Semantic Web Databases. In: Business Intelligence for the Real-Time Enterprise. volume 206 of LNBIP. Springer, pp. 81–96, 2013.

[Mo14] Morton, Kristi; Balazinska, Magdalena; Grossman, Dan; Mackinlay, Jock: Support the Data Enthusiast: Challenges for Next-generation Data-analysis Systems. PVLDB, 7(6):453–456, 2014.

[NL12] Nebot, Victoria; Llavori, Rafael Berlanga: Building Data Warehouses with Semantic Web Data. Decis. Support Syst., 52(4):853–868, 2012.

[Ru14] Rudolf, Michael; Voigt, Hannes; Bornhövd, Christof; Lehner, Wolfgang: SynopSys: Foundations for Multidimensional Graph Analytics. In: Business Intelligence for the Real-Time Enterprise. volume 206 of LNBIP. Springer, pp. 159–166, 2014.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 61

Reverse Engineering Top-k Join Queries

Kiril Panev[1], Nico Weisenauer[2], Sebastian Michel[3]

Abstract:

Ranked lists have become a fundamental tool to represent the most important items taken from a large collection of data. Search engines, sports leagues and e-commerce platforms present their results, most successful teams and most popular items in a concise and structured way by making use of ranked lists. This paper introduces the PALEO-J framework which is able to reconstruct top-k database queries, given only the original query output in the form of a ranked list and the database itself. The query to be reverse engineered may contain a wide range of aggregation functions and an arbitrary amount of equality joins, joining several database relations. The challenge of this work is to reconstruct complex queries as fast as possible while operating on large databases and given only the little amount of information provided by the top-k list of entities serving as input. The core contribution is identifying the join predicates in reverse engineering top-k OLAP queries. Furthermore we introduce several optimizations and an advanced classification system to reduce the execution time of the algorithm. Experiments conducted on a large database show the performance of the presented approach and confirm the benefits of our optimizations.

Keywords: data exploration, reverse query processing

1 Introduction

Data in modern relational database systems and specifically in data warehouses is often stored in complex schemas. In order to write analytical queries, database users need to understand the entire schema, which limits the use of the systems to expert users already familiar with the data. Different research areas are working on developing user-friendly ways for querying and exploring data. In this work, we propose an approach of exploring database contents by means of ranked lists. Rankings often capture the essence of the available data, they represent a small subset whose characteristics are important to investigate. In our approach, a user submits a ranked list of entities with their scores and the system returns a set of queries that when executed over the database generate results that match the user's list. We identify OLAP-style select-project-join queries with arbitrary number of joins. There are various important application scenarios. Consider, for instance, business analysts who want to find the different factors that yield some entities of interest as top-performing. By inspecting an identified query, they will learn about the common characteristics of the top entities and about the way the data is organized in the underlying database. Furthermore, more efficient queries wit less-complex join condition can also be found as alternatives. Previous work [Zh13, Sh14, Ps15] considered the problem of reverse engineering join

[1] TU Kaiserslautern, Germany, panev@cs.uni-kl.de
[2] TU Kaiserslautern, Germany, n_weisenau10@cs.uni-kl.de
[3] TU Kaiserslautern, Germany, michel@cs.uni-kl.de

Customer

CustID	Name	Country	Balance
1	Bruce Wayne	USA	250.49
2	Clark Kent	USA	124.56
50	Tony Stark	USA	45.99

Part

PartID	PName
1	LG G4
2	Galaxy Note
3	iPhone 7

Orders

OrdID	CustID	Price	Date
1	1	199.99	28/11/14
2	1	749.90	01/04/15
23	2	199.99	30/12/12
93	50	1000.00	27/02/15

LineItem

ItemID	OrdID	PartID
1	1	1
2	2	2
15	23	1
18	50	3

Fig. 1: Sample database with sales data

Tony Stark	1000.00
Bruce Wayne	749.90
Clark Kent	199.99

(a) Input L

```
SELECT c.Name, MAX(o.Price)
FROM Customer c, Orders o
WHERE c.CustID = o.CustID
AND c.Country = 'USA'
GROUP BY c.NAME
ORDER BY MAX(o.Price) DESC LIMIT 3
```

(b) Result query

Fig. 2: Example input L and result query

queries, however none of them support OLAP-style queries with aggregation functions nor are based on a ranked top-k input.

Figure 1 shows a sample database containing sales data. It contains information about customers, like where they come from, the orders they placed, what parts were bought in each order etc., as in TPC-H [TP]. The arrows denote the direction of foreign key to primary key relationship between the tables. In this scenario it is very common to join the relations to gain insight about the ordering patterns of customers.
Now, consider the top-k list shown in Figure 2a. It contains the customer name and the score that was used as ranking criteria according to which the customers are ordered. Considering the database in Figure 1, we can find that the top-k list can be generated using the query in Figure 2b. It computes the top-3 customers, living in the USA, ranked by the maximum price of an order that they made. There can be other queries that produce the same input list, which enables further exploration of the underlying data. Reverse engineering even this rather simple top-k query on the sample database is not trivial, as the information that can be derived from a two-dimensional top-k list is very limited, especially with small k.

1.1 Problem Statement

Given a database D with tables $\{T_1, T_2, \ldots, T_t\}$ with schema $T_i = \{A_{i_1}, A_{i_2}, \cdots\}$, and an input list L that represents a ranked list of items with the numerical values that were used for the ranking. The system task is to efficiently and effectively discover queries that, when executed over the database D, compute result lists that match the input L.

```
SELECT L.e, agg(value)
FROM R1, R2, ...
WHERE P1 and P2 and ...
GROUP BY L.e
ORDER BY agg(value) DESC
LIMIT k
```

(a)

L	
$L.e$	$L.v$
e	100
f	90
g	80
m	70
o	60

(b)

Fig. 3: Query template (a) and example input L (b)

We identify top-k select-project-join queries of the form shown in Figure 3a. This task can be broken down into three sub-tasks: identifying the join predicates, finding the filter predicates and ranking criteria, and validating the queries. In this work, we specifically focus on finding the query graph, i.e., identifying the foreign key/primary key constraints of the query. We abbreviate primary key as pk, and foreign key as fk in the remainder of this paper.

Definition 1 *A **query graph** is a graph Q with nodes corresponding to database tables in D and edges representing fk/pk constraints.*

The task is to efficiently identify the join constraints of a valid query, i.e., one whose result matches L. This consists of determining the tables in the FROM clause of the query (e.g., Customer c, Orders o), as well as the fk/pk relationships therein (e.g., c.CustID = o.CustID). Identifying filtering predicates and ranking criteria was discussed in prior work [PM16] where the system operates on a single table T and joins were not considered.

The ranked list L, which serves as input for the system and resembles the desired query output, consists of two columns and k rows. The first column, denoted $L.e$, contains entity identifiers, while the second column, denoted $L.v$, contains possibly aggregated numerical values. The input is ordered by the second column, meaning the top-k entities are ranked based on their "score". L does not contain any information about the database columns containing the entities or the aggregated score values. An example input list is shown in Figure 3b; the column names are omitted in the input as they need not generally be related to database column names.
In addition to finding the fk/pk join constraints, we support predicates P of the form $P_1 \wedge P_2 \ldots \wedge P_m$, where P_i is an atomic equality predicate of the form $A_i = v$ (e.g., Country = 'USA'). We denote with *size* of a predicate $|P|$ the number of atomic predicates P_i in the conjunctive clause.

1.2 Contributions and Outline

In this work, we present an approach that specifically deals with solving the task of reverse engineering top-k *join* queries with arbitrary number of fk/pk joins. With this paper we make the following contributions:

- We present the PALEO-J framework which efficiently computes a list of join query candidates by making use of database meta-data and appropriate data structures.
- Additionally, we present a ranking and verification system which ensures fast execution of candidate join queries. Furthermore, queries with a high probability of being a match are tested first while keeping database interactions low.
- We introduce several optimizations which improve the running time of the framework, including a candidate classification system.
- We present the results of experimental evaluation conducted on a database containing TPC-H [TP] benchmark data.

The paper is organized as follows. Section 2 discusses related work. Section 3 presents system overview. Section 4 establishes the key ideas of our approach to handling joins, followed by introducing certain optimizations in Section 5. Section 6 gives a brief overview of the remaining parts of our framework, while Section 7 reports on the results on the experimental evaluation and presents lessons learned. Section 8 concludes the paper.

2 Related Work

Reverse engineering queries was considered by Tran et al. [TCP09, TCP14] in a data-driven approach which models the problem of finding instance-equivalent queries as a data classification task. This dynamic class-labeling technique is built on a decision tree classifier where nodes are split based on the Gini index, indicating their contribution to the desired query output. While this approach is useful to discover instance-equivalent queries which may be a simpler representation of the original query or help to understand the database schema better, it cannot be used to discover complex OLAP queries given a top-k input list.

Sarma et al. [Sa10] consider only one relation without joins and projections. They focus on finding the selection condition and consider it as an instance of the set cover problem [Va01]. They present separate approaches for different types of queries and we are interested in conjunctive queries with any number of equality predicates. Their proposed algorithms are rather limited and utilize the size of the attribute domains in the view.

Psalidas et al. [Ps15] study the problem of discovering project-join queries which *approximately* contain a given set of example tuples to ease the process of understanding the database schema and writing a specific query. They generate the top-k output list of suitable queries and do not support top-k queries. The systems relies on offline building of large indices and introduce a spreadsheet-style keyword search system based on candidate-enumeration and candidate-evaluation with a flexible caching component. Their scoring model allows to tolerate relationship and domain errors with the given example tuples to make up for human errors while still providing a relevant top-k list of queries.

Shen et al. [Sh14] focus more on the verification of candidate queries and do not consider selection. The system also takes a set of example tuples as input, but tries to discover the

minimal project-join query containing the exact set of example tuples in its output. The system requires the database tables to contain only textual data to apply the approach of pruning many invalid candidate queries at once by applying so called *filters*, which resemble the main contribution of their work. The filters exploit subtree relationships between the candidates and rely on full-text-search indices on each text column in the database.

Li et al. [LCM15] propose finding queries from examples, to help non-expert database users construct SQL queries. The user first provides a sample database D and an output table R which is the result of Q on D. The proposed approach is able to identify the user's target query by asking for the user's feedback on a sequence of a slightly modified database-result pairs, which are generated by the system. The authors use the work of [TCP09] for generating candidate queries and focus on optimizing the user-feedback interactions to minimize the user's effort in identifying the desired query.

Reverse engineering complex join queries is studied by Zhang et al. [Zh13]. Their main insight is that any graph can be characterized as a union of disjoint paths, connecting its projection tables to a center table, called a *star*, as well as a series of merge steps over this star table. Using this insight, the proposed algorithm filters out candidate queries that need not be tested. To keep track of the candidates, a lattice structure is used where each vertex represents a star and the edges represent merge steps. While this system allows to efficiently discover complex join queries, it is not able to handle top-k OLAP queries containing aggregations or selections. We use their approach as guidance, however using a small top-k list as input and supporting aggregation functions change the problem significantly.
None of these systems allow the reverse engineering of complex top-k OLAP queries including aggregations and fk/pk joins based on a ranked top-k input.

Reverse engineering query processing is studied in [BKL07,Bi07,BCT06,MKZ08], however their objectives and techniques are different. Binning et al. [BKL07, Bi07] discuss the problem of given a query Q and a desired result R, generate a test database D such that, $Q(D) = R$. Bruno et al. [BCT06] and Mishra et al. [MKZ08] consider generating test queries to meet certain cardinality constraints on their subexpressions.

3 System Overview

The system for discovering top-k SPJ queries with arbitrary number of joins contains three main components. We see the three main components depicted in Figure 4.

- Identify the join predicate, i.e., what are the possible fk/pk constraints in the query
- Identify the filter predicates and ranking criteria
- Validate queries

The first component, resembling the contribution of this work, consists of generating a ranked list of candidate joins by using mostly database meta-data and minimal access to actual table data.

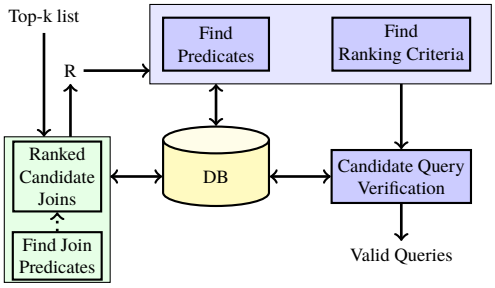


Fig. 4: System architecture

Definition 2 *We say a query graph Q with a set of tables T_i and their fk/pk join predicates is a* ***candidate join*** *iff projecting and selecting on the table R created from the join can result in the input list L. Formally,*

$$R = \bowtie (T_i \in Q), \;\; \exists \;\; projection \;\; \pi, \; selection \;\; \sigma \;\; : \;\; \pi(\sigma(R)) = L$$

The table R created from a candidate join contains all the tuples from the input list L. Furthermore, it contains additional tuples that are not (yet) filtered out by a filter predicate and other columns that will not be used by the projection. Additionally, it contains multiple tuples per entity from which the ranking criterion needs to compute an aggregated score. Thus, the table R is a superset of the input list L and contains all the necessary data to produce the input list by applying the appropriate filter predicates and ranking criterion.

The second component takes as input the candidate join predicates and, for each created join table R, identifies possible filter predicates and ranking criteria. Then, by combining the join conditions, the filter predicates, and the ranking criteria, full-fledged candidate queries are created. The final component verifies these queries by executing them on the database and comparing their results with the input, those that match the input are returned as valid queries. The latter parts of our system are briefly reviewed in Section 6. In this paper, we devise an efficient approach in identifying the join predicates and consider techniques that could facilitate the next steps to determine the remaining parts of a valid query.

4 Finding Join Predicates

The task of identifying candidate join trees consists of the steps depicted in Figure 5. These steps are all part of the Find Join Predicates component, shown in the lower-left part of Figure 4. The following sections will describe the six steps needed to generate the ranked candidate joins from the input L, which will then be processed by the remaining components of the framework.

4.1 Step 1: Schema Exploration

In the Schema Exploration step, the system tries to determine which column in the database schema was used as the entity column. Furthermore, by leveraging basic database metadata, it selects suitable columns as early candidates for the ranking criteria.

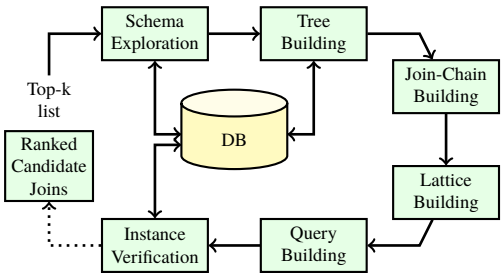


Fig. 5: Generating the ranked candidate joins

Identifying the correct entity column is a straight-forward task. Finding the column from which the input entities originated is done by using a B+ tree index, or alternatively, the entity column can be efficiently identified by implementing an auxiliary inverted index. Finding suitable ranking criteria, however, cannot be done by simple inclusion check of the values in L, since these may stem from an aggregation. Thus, candidate ranking columns are determined based on the data type of the columns.

4.2 Step 2: Building Join-Trees

For each entity candidate column $T.e$, this step explores the database schema and tries to build a tree-like query graph that we call a join tree rooted at $T.e$. All candidate joins are extracted from such trees. We define a depth d of the tree that limits the complexity of the join candidates inspected by the following stages of the framework. A sufficiently high depth d guarantees the successful discovery of a matching query generating L. In turn, if d is chosen too high, the performance of the algorithm may suffer, as a large amount of overly complex candidate joins will be generated.

An example jointree of depth 2 is illustrated in Figure 6 following the TPC-H schema, with nodes corresponding to instances of database tables and edges relating to key-constraints between tables. Starting from the entity candidate table Customer as a root node, the child table Nation is referenced by a fk inside the Customer table, while the Orders child node contains a fk referencing the pk of the Customer relation. All tables related by fk/pk keys will be added as child nodes to the current node, until the maximum depth of the tree has been reached. Hence, each node except for the root node contains its parent node as a child node, unless it is a leaf of the tree. As this results in a lot of nodes instantiating the same database relation, a running index is appended to each node identifier, which now consists of the table name and a number (e.g., Customer1). Having multiple instances of the same table in a join can serve as a proper filter and removing one of them changes the output. Unique node identifiers are important for referencing nodes in later steps of the algorithm, especially when nodes are being merged. Additionally, all of the edges carry information about the table and column names involved in the join to ease the translation into SQL during the Query Building step. The process of building the tree does not have a large performance impact on the database, as the database metadata needs to be queried only once to gain information about all key-constraints in the relevant schema.

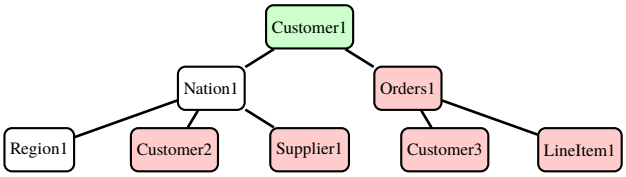


Fig. 6: Example tree of depth 2 with marked nodes

Marking nodes in join trees. After a tree is built, the algorithm continues with marking certain nodes. All ranking candidate columns which have been identified during Schema Exploration will be looked up in the tree and nodes originating from the corresponding tables will be marked as ranking candidate nodes $T.r$, which may also include the root of the tree. Figure 6 shows an example tree with ranking candidate nodes marked in red. The root node of the tree is both an entity and a ranking candidate and therefore marked with green in the figure. Marking the nodes helps to simplify the approach of constructing the join chains, which always have to include at least one marked node and the root of the tree.

4.3 Step 3: Building Join Chains

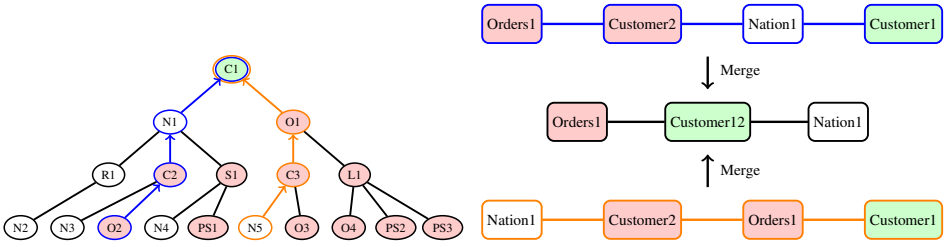
The goal of building the join chains and the lattices during the following step is to represent all possible query graphs connecting the entity table to a table containing a ranking candidate column. Hence a join chain is a set of nodes from the join tree built in Step 2 which includes the entity node and at least one marked node.

Definition 3 *A **join chain** $\mathcal{J}$ is a linear query graph that always contains the database table T_e which was identified as the entity table during Schema Exploration as one node of the graph and at least one node corresponding to a table T_r which contains a ranking candidate column.*
A join chain $\mathcal{J}$ generalizes a query graph Q if the following two conditions are true:

1) *The set of tables corresponding to nodes in Q is equal to the set of tables corresponding to nodes in $\mathcal{J}$*
2) *Each node in Q can be mapped to a node in $\mathcal{J}$ and this mapping is an injective function relating nodes to nodes corresponding to the same database table.*

To construct a join chain, the method begins with a random node in a join tree $\mathcal{T}$, which serves as the starting point of the join chain, and then navigates to its parent node, adding it to the chain in the process. This step is repeated until the root of the tree has been reached, concluding the join chain construction by adding the entity node T_e. By consecutively using all individual nodes contained in the tree as a starting point for a join chain, all possible chains are eventually constructed.

Figure 7a shows an example join tree of depth 3. Two join chains are highlighted in the tree, one starting at the node O2, and the other starting at N5. Both join chains essentially contain the same set of tables: two instances of the Customer relation and one instance each



(a) (b)

Fig. 7: Example join tree (a) and merging of nodes (b)

of the Orders and Nation relation. Hence, these join chains only differ in the order of the join operations, the reason for this being the multiple instances of the Customer relation.

4.4 Step 4: Building Lattices

Join-chains are linear graphs connecting nodes of a tree to its root. The join chains generated during the previous step will serve as root nodes for the lattices built in this next step of our approach. By iteratively merging duplicate nodes, i.e., nodes stemming from the same database table, new query graphs are built, corresponding to vertices inside the lattices, as shown in Figure 8. Each edge in the lattice translates to one merge step. By performing these merge steps over all join chains, all possible query graphs will be built eventually, meaning the query graph representing the query generating L will be among them if the depth of the computation was chosen sufficiently high.

Theorem 1 *If the data inside the database has not changed, then for any query graph Q, there exists at least one join chain $\mathcal{J}$ that generalizes Q, and Q is a node in the lattice rooted at $\mathcal{J}$.*

Theorem 1 is inspired by Section 3 in [Zh13] and its proof follows the one presented there.

Merging nodes. Theorem 1 assures that the query graph of the query which generated L is among the candidate graphs after building the lattices. By merging nodes representing identical database tables, the complexity of the individual graphs decreases with every merge step performed, as the number of nodes inside a graph is reduced.

Unless the depth of the tree that the join chains originated from is very low, there will be a number of nodes referencing identical database tables. This leads to multiple merge steps that might be needed to generate a fully merged graph, as seen in Figure 8. In this situation it is important to maintain the unique identifiers of merged nodes during intermediary merge steps to be able to distinguish nodes, as they could be mapped to the same identifier after performing pairwise merging.

The process of generating a set of lattices starts with adding the root of the lattice and performs a merge operation for each possible pair of nodes referencing identical database

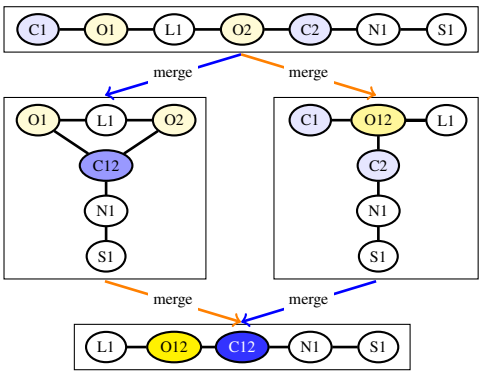


Fig. 8: Example lattice

tables, which will result in a row of new graphs. To successfully merge a pair of two nodes, the resulting merged node retains the union of edges and neighbors of the two original nodes. To indicate that the resulting node is a product of a merge step, it will have a modified identifier which consists of the table name and the two number identifiers of the original nodes appended to each other. After this first iteration, the method continues with the graphs resulting from the first merge step and performs further merging to expand the lattice with new graphs until all possible merge steps have been performed and all graphs have been inspected.

When adding new graphs to the lattice, the lattice keeps track of the number of merge steps needed to generate the newly added graph and keeps graphs with the same amount of merge steps in the same set. This allows us to establish the lattice structure with the initial join chain at the top and the fully merged graph with no nodes referencing identical tables at the bottom.

Handling duplicate graphs. The two join chains visualized in Figure 7a are identical with respect to the set of referenced database tables, and both contain two instances of the Customer relation. Figure 7b shows that both of these join chains collapse to the same chain after performing a merge step on the two Customer nodes. The resulting graph connects the entity candidate Customer relation to the ranking candidate Orders, but also retains the join between the Customer and the Nation relation. The join with the unmarked Nation node appears to not have an impact on the result of the query, as it does not connect the entity table to a ranking candidate, but it may still have an influence on the outcome of a query utilizing predicates to filter Customer entities based on attributes of the Nation relation. Thus, it is important to retain the unmarked nodes, even when they are "dangling" nodes as shown in the example.

After merging the nodes in the two join chains of the example, one of the result graphs can be eliminated due to being a duplicate, therefore reducing the overall number of candidates. Keeping track of duplicate graphs is essential to reduce the computation time of the upcoming steps, which require database interactions and therefore represent a bottleneck of the overall computation.

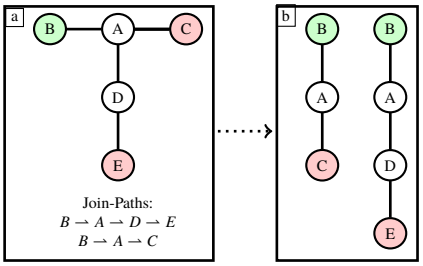


Fig. 9: Join path representation of different query graphs

4.5 Step 5: Query Building

Converting a query graph into an SQL query requires extracting various pieces of information to form the SELECT, FROM, and WHERE clause. The query graph stores this information and facilitates the construction of SQL queries.

Each join-query gets assigned a cost value that is used for prioritizing the least costly candidate joins to be tested first. We use the estimated size of the join-result table R as an indicator of how expensive is a candidate join. The relative size of R can be calculated by a simple formula as proposed by Swami and Schiefer [SS94].

$$|T_1 \bowtie T_2| = \frac{|T_1| \times |T_2|}{max(d_1, d_2)} \tag{1}$$

where d_1 and d_2 are the cardinality of the join-columns in T_1 and T_2, respectively.

The queries are then sorted in ascending order of their cost and handed to the next step of the computation, the Instance Verification, which aims for eliminating some of the query candidates before the final candidate list is passed on the next components of the framework.

4.6 Step 6: Instance Verification

During Instance Verification the scores present in the input list L will be compared to actual values in the database columns to rule out candidate joins which do not contain suitable ranking columns. Since the database interactions needed for this step are expensive, it is important to make use of an efficient verification approach which keeps the number of tuples involved in the database operations to a minimum.

Queries may contain joins with tables which are not connecting the entity table to a table containing candidate ranking columns. The goal of Instance Verification is to eliminate candidate ranking columns which cannot generate the input scores. Hence, our goal is to represent each candidate join as one or multiple paths between the table containing the entities and the tables containing the ranking columns.

Figure 9 displays candidate joins represented by their query graphs, the entity table is marked in green while the tables containing ranking candidates are marked in red. The first, shown in Figure 9a, can be represented by the two paths $B \rightharpoonup A \rightharpoonup D \rightharpoonup E$ and $B \rightharpoonup A \rightharpoonup C$.

If a query graph can be represented by two or more paths of various lengths, it usually shares those paths with less complex queries that have already been covered by Instance Verification, as the arriving queries are ordered by their estimated cost. Two less complex queries are shown in Figure 9b, each representing one join path of the query in Figure 9a. By reusing results of already processed queries we can speed up the verification of more complex queries.

To generate the join paths used by Instance Verification, Dijkstra's Shortest Path Algorithm [Co01] is used on the query graphs. Dijkstra's Algorithm will calculate the shortest distance, meaning the least amount of joins needed, from the entity node to each of the ranking nodes, which will result in the respective join paths being built.

The prefix-path relationship. One important assumption about the OLAP-queries generating the input list L is that all joins which are not connecting an entity table to a ranking table do not increase the number of tuples used for score aggregation, hence they are used for applying filters. Otherwise, one would aggregate over duplicate values in the ranking column, resulting from joining the ranking table with another table in pk/fk direction.

Having calculated a join path for each candidate join query, we can use them to reduce the computation time of the actual Instance Verification. This can be achieved by keeping track of the validity of previously verified queries. If a valid candidate query's join path is a prefix of another query's join path, that query can also be considered a valid candidate. This is the case because the less complex query already contains suitable ranking columns, which are also present in the more complex one. This complex query may apply further filtering or not, hence making it a valid candidate query in any case as the boundary conditions for the ranking columns stay the same. In the opposite situation, where a non-valid candidate's join path is a prefix of another query's join path, the algorithm can make use of this result by not testing the already invalidated columns again. For queries which do not share a prefix with an already verified query we can make no assumptions and therefore a full verification needs to be performed.

Candidate join query validation. By default Instance Verification takes the top entity from the input L and evaluates the previously computed join path on the database by executing the respective SQL-query. It is possible to consider more entities from L, resulting in a higher computation time, but also with the possibility of eliminating more ranking candidates.

The result of joining the top entity along the join path is a verification relation V, which contains all possible ranking columns. In order to be a valid ranking candidate, a column has to be able to generate the score in L corresponding to the entity used for generating V. When aggregating the values in a ranking candidate column, all tuples corresponding to the input entity will be used to generate this aggregated score. Therefore this score cannot

be directly compared to the score in L, since the aggregation generating the input score may have used only a subset of those tuples, as additional predicates may have filtered some of them out. Hence, score boundaries can only be defined by utilizing the SUM- and MIN-aggregation functions on the columns of this table.

Figure 10a depicts the decision tree showing how a query will be classified as either a valid or non-valid candidate join. The values of the columns being considered are denoted as c_i and c_j. Note that for a query to be valid candidate, only a single column or column pair needs to be classified as valid, whereas for the query to be non-valid, all columns and column pairs need to be classified as non-valid.

The final list of valid candidate queries is ranked by the calculated cost value and handed to the next component of the framework for discovering possible filter predicates and the aggregation function.

5 Optimizations

In this section we propose an advanced classification system to improve the ranking of the candidate join trees, along with other optimizations to decrease the running time of the extended framework.

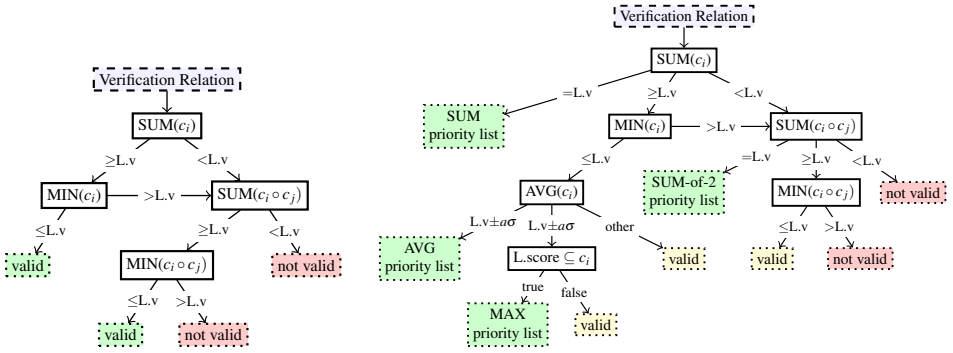
The ranking of candidate join queries has the most influence on the overall running time of the algorithm, as each candidate query has to be executed on the database to compare the actual results to the given input list. In case of long-running queries that contain many cost-intensive joins of large tables, each execution of a candidate query is very time consuming. Our goal is therefore to find the correct query with the first candidates joins that are handed to the rest of the framework, reducing the amount of mismatches to a minimum.

5.1 Improved Query Ranking

The improved query ranking is achieved by identifying possible aggregation functions and columns during the Instance Verification step and by putting the respective queries into higher candidate priority lists. During the identification of the specific aggregation functions, one has to be aware of their particular characteristics which may result in varying precision of identifying the individual functions.

Queries using either the AVG- or the MAX-aggregation function can be identified more precisely than queries using a SUM-aggregation, since the former two aggregations' output values lie within the range of a column's original values. Furthermore, the MAX-aggregation can be identified more reliably than the AVG-aggregation.

After building the priority lists, the candidate join queries inside will be ranked again based on their cost value. The aforementioned lists not only contain the queries which will most likely lead to a matching result query, but also carry information about the aggregation functions which should be considered in the next steps of the framework. To achieve



(a) Basic (b) Optimized

Fig. 10: Classification of candidate queries

the improved query ranking, more database interactions are needed during the Instance Verification step, which results in a certain overhead being added to every execution. Instead of only considering the sum and the minimum of the values inside a column, now also the mean value and the standard deviation are being computed. This additional information will be used by the algorithm in conjunction with further heuristics to build the priority lists. The improvement in the query ranking outweighs the overhead and we show this in the experiments.

We introduce heuristics that can be applied to build the priority lists, each for the different ranking criteria supported by our system. If the score of an entity lies within a-times the standard deviation of the mean value of the column, then the candidate query containing this column will be entered into a certain priority list. The parameter a should be set sufficiently low, so that few false positives occur, but also high enough to exclude false negatives. Experiments have shown that for the different ranking criteria, a specific margin of the standard deviation fulfills this requirement for most queries.

5.2 Advanced Classification of Queries

The generation of priority lists, and therefore the improved query ranking model, makes use of an advanced classification tree, which is illustrated in Figure 10b. Each of the previously mentioned heuristics for identifying candidate ranking criteria is present inside this classification tree, along with the corresponding priority lists.

A condition containing $L.v$ refers to the score value of the input entity, while $L.score$ refers to a set of score values of the input list L. Note that it is possible to follow multiple edges when checking the conditions in the decision tree, as these are not necessarily mutually exclusive. However, we assume the conditions associated to the edges are checked from left to right and therefore follow a strict order. If the query is not put into any of the various priority lists, it will always be categorized as either *valid* or *not valid* analogous to the baseline Instance Verification approach.

Fast Verification. If a candidate query has been categorized into one of the priority lists, it is very likely to result in a match. Hence the latter parts of the framework should begin processing this query and all related queries as soon as possible. In order to speed up the remaining Instance Verification process, this optimization called Fast Verification allows to skip through the remaining candidates if at least one query has already been put into a priority list.

During Fast Verification, for the remaining candidates, only the join-paths will be inspected. If a join-path of a query inside the priority list is a sub-path of a path of the currently inspected candidate, this candidate will also be put into the same priority list. Otherwise this candidate join query will not undergo any further inspection and will be added to the list of valid join candidates. Thus, if no matching query was found by processing the candidate joins inside the priority lists, such a join query will be handed to the finding filtering predicates component.

6 Discovering Filter Predicates and Ranking Criteria

The techniques for discovering filter predicates and ranking criteria are extensively discussed in [PM16]. The approach there, considers working on a single input table R, which can now be generated in advance by executing the valid candidate joins. As the result of Instance Verification is a ranked list of join-query candidates, they can be executed in order of their rank to generate the table serving as input for the remaining parts of framework, as shown in Figure 4.

To keep the memory usage low, these tables are filtered to only contain the entities from the top-k input list. We denote the joined table that contains only tuples from the input entities as R' and store it in-memory. The system first identifies a set of *candidate predicates* which can be of the form $P_1 \wedge P_2 \ldots \wedge P_m$, where P_i is an atomic equality predicate of the form $A_I = v$ (e.g., country = 'Sweden'). For each candidate predicate there must exist at least one tuple t_i for each entity in L with $P(t_i)$ assessing to true.

A naïve approach would take all possible predicates as candidates and proceed to finding ranking criteria. This would significantly reduce performance, since the expansion of the set of candidate predicates would explode. The system utilizes an apriori-style algorithm that uses the anti-monotone property of the criteria what is considered a candidate predicate. It starts by identifying atomic candidate predicates with size $|P| = 1$ and iteratively creates larger conjunctive predicates by efficiently using the ones created in the previous step.

In order to identify suitable ranking criteria, the system uses the set of determined candidate predicates. It utilizes small data samples, histograms, and simple descriptive statistics which are computed upfront, thus selecting suitable columns and aggregation functions as ranking criteria and avoids touching actual table data. Queries are created by combining the selected ranking criteria with the candidate predicates which are efficiently verified on R'. The queries that produce results that match the input list L are marked as candidate queries, which because of the possible presence of false positives [PM16], need to be verified on the base table that contains *all* tuples, not only tuples with the input entities.

Parameter	Value
depth d	5
minJoins	0
maxJoins	3
maxQueries	10

Tab. 1: Parameters used in the experiments

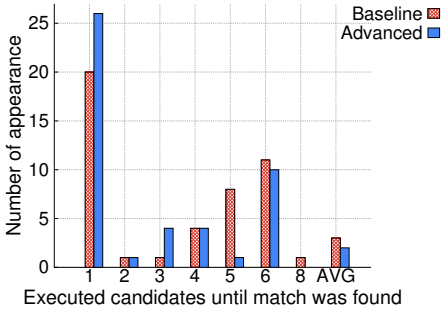


Fig. 11: Number of executed candidate queries

In the last component of our framework, the candidate queries are evaluated using the base table data. Unlike the Instance Verification step presented in Section 4, where the focus is whether the joined tables *are suitable* to produce the input list, here full-fledged candidate queries are executed and their results are compared if they *match* the input list. The queries that match the input list are returned to the user as valid queries.

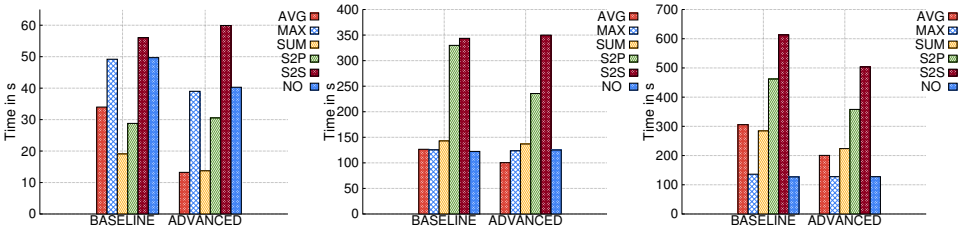
7 Experimental Evaluation

The implementation of our framework was done in Java. The experiments have been conducted on a machine running Ubuntu Linux 14.10, powered by two Intel Xeon E5-2603 hexa-core CPU at 1.60GHz, with 128GB of RAM, using Oracle JVM1.8.0_45 as the Java VM (limited to 20GB memory). The tables are stored in a PostgreSQL 9.4 database, with B+ tree indexes created on the possible entity columns.

Dataset: The evaluation of our system was done using the TPC-H [TP] benchmark. We created a a 10GB dataset, i.e., a scale factor 10 instance of the benchmark.

Input lists: The input used for the experiments consists of 46 query outputs. The queries used to generate the outputs make use of varying scoring functions, between one and three join operations and between zero and three predicates.

All experiments have been conducted with the parameter values shown in Table 1. The depth d for generating the tree in Step 2 of the computation has been set to 5, queries with zero to three joins will be considered as candidates and the maximum amount of candidates that will be considered for finding filtering predicates and ranking has been set to 10. These values can be chosen based on the database schema complexity and the estimated complexity of the query to be reverse engineered and allow to limit the execution time of the algorithm by not generating an unnecessary large amount of candidates. Tuning parameter a has been set to reasonably small values to allow proper categorization into the priority lists. There were three queries that were outliers, executing their candidate queries on the database took significantly more time than any of the remaining queries in the workload. In order not to skew the results, we do not include these queries in the presented results.



(a) 1-Join (b) 2-Joins (c) 3-Joins

Fig. 12: Average time to find queries with different query graph size $|Q|$

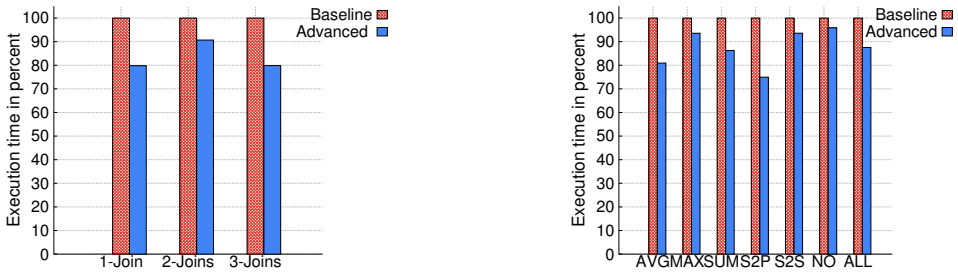
If the basic PALEO-J framework without any optimizations was used to run the experiments, we refer to it as the *baseline approach*, otherwise, we refer to the *advanced approach* when all of the optimizations were utilized. Note that in the evaluation we focus on identifying the join conditions of the query. Finding filtering predicates and ranking criteria is discussed in depth in our previous work. Furthermore, the reported results focus on efficiency of discovering the first valid query in the results that are presented.

Finding a matching query. First, we want to point out that all of the 46 queries have been successfully reverse engineered by the PALEO-J framework, regardless of whether the baseline approach or the advanced approach was applied.

The goal of the optimizations of the advanced approach is to improve the ranking of the candidate join queries, which are given as input to the next steps of the framework. The optimal ranking has the query which is most likely to be a match at the top of the candidate list. Therefore, we show the quality of the ranking by inspecting how many candidate joins have to be executed until a matching query is found.

Figure 11 shows this statistic for the two presented approaches. On average, the advanced approach inspects approximately two queries to find a match while the baseline approach needs about three inspections. The baseline has many cases of inspecting five queries before finding a match and even has a single case where eight inspections were needed. Depending on the time needed to inspect a single candidate query, the difference between inspecting three or two candidate queries on average may be significant, as the following experiments show.

The process of finding a matching query consists of the six steps to generate the candidate list and further finding filtering predicates and ranking criteria by the framework until a matching query is discovered. Figure 12 displays the average time until a matching query is found, grouped by the different scoring functions supported by our framework and the number of joins. While data labeled with AVG, MAX and SUM directly stems from reverse engineering queries using the corresponding aggregation functions, S2P and S2S refers to queries using the *sum of the product* or the *sum of the sum* of two columns. Finally, the label NO corresponds to data stemming from reverse engineering queries with no aggregation function.



(a) By number of joins

(b) By scoring function

Fig. 13: Relative execution times

Figure 13a compares the relative execution times of the advanced approach to the average running time of the baseline as the reference point. On average the advanced approach is faster in reverse engineering a query regardless of the amount of joins and it performs 10–20% faster than the baseline approach.

Figure 13b not only groups the results by scoring function instead of the number of joins, but also averages over the **relative** execution times of reverse engineering individual queries. This is done to make the results between the queries with a different amount of join operators comparable, since they heavily vary in the absolute time needed to find a match. Without this adjustment, the results of reverse engineering queries with three joins would dominate the overall result.

As can be seen in Figure 13b, reverse engineering queries with the MAX-aggregation function or no aggregation function profits the least from the optimizations of the advanced approach, with only about 5% decrease in running time averaged over the amount of joins. The S2P queries profit the most and are discovered about 30% faster. Overall, queries are found almost 14% faster on average when using the advanced approach, ignoring the amount of joins and the used scoring function.

Generating the candidate list. To gain insight into the execution time of the six steps of finding the join predicates, Figure 14 shows the median execution times of these steps based on the 46 test queries.

Steps 1–5 do not vary between the baseline and the advanced approach and also remain relatively constant between the individual queries. Among the first five steps, Step 1 and 2 take the most time, as they are interacting with the database to gain information about the schema. The added up execution times of Steps 3–5 range in the milliseconds even though they are dealing with the more complex data structures, which shows the impact of even the smallest database interactions. The Instance Verification step differentiates the baseline from the advanced approach and the experiments show a significantly increased execution time in the latter, due to the extended classification process. However, this is a small cost to pay compared to the benefit that is gained in the next steps. Looking only at the six steps to generate the list of candidate join queries, we can say that the candidate joins are found within a few seconds for both the baseline and the advanced approach.

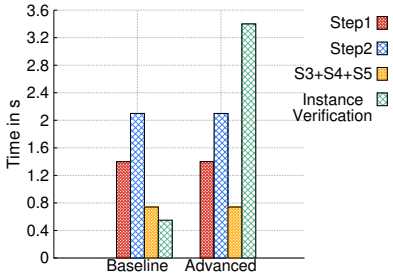


Fig. 14: Execution times of the join-framework steps

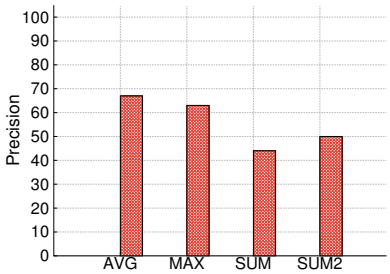


Fig. 15: Precision of categorizing the result query into the correct priority list

Precision of the advanced approach. The improved ranking of the advanced approach can only be achieved if the candidate query which resembles the query generating the input list L is categorized into a priority list for early inspection. To gain insight into the accuracy of the advanced classification process, we can check how often candidate queries are put into the priority lists which correspond to the matching queries' scoring function.

Figure 15 shows the precision of this classification grouped by the different priority lists. When comparing this statistic to the one showed in Figure 13b, one can see that the accuracy of the classification does not correlate with the actual performance gain of making use of the improved ranking. This is the case because each aggregation function has a different impact on the quality of the candidate ranking of the baseline approach. The MAX-aggregation seems to result in a good ranking of query candidates when using the regular Instance Verification, while the advanced Instance Verification step invests a lot of time into checking columns for the inclusion of several input scores to put a candidate into a MAX priority list, resulting in a large overhead.

Overall 50% of the scoring functions involving two columns and an average of almost 70% of the scoring functions involving a single column have been correctly identified during Instance Verification of the advanced approach.

Lessons learned. Both the baseline and the advanced approach were able to reverse engineer every single query of the 46 input queries. The advanced approach, which makes use of the advanced classification system, executed fewer candidates before it found the matching query. The advanced classification system also comes with an overhead. This overhead pays off, since the performance gain from improving the ranking of the candidate list depends on the time needed to execute the individual candidates, which are expensive because of the present joins.

The advanced approach always outperforms the baseline and more complex join queries have bigger performance gain. Having more join operations leads to slower candidate query execution, thus the ranking played a bigger role when reverse engineering these complex queries. The advanced approach reduced the time needed for reverse engineering join queries by up to 20% compared to the baseline approach.

8 Conclusion

We presented the PALEO-J framework for reverse engineering top-k database join queries. This is a challenging problem, given the small input and the amount of database tables that have to be considered as potential join partners due to the lack of information to filter candidates out. Having only entities and aggregated scores to find and filter out candidates, we had to find a way to interact with the database system to efficiently classify the large amount of candidate queries. This was achieved with the proposed Instance Verification step, which was refined with the advanced classification system to significantly improve the query ranking. The introduction of priority lists improves the ranking of candidate join queries, which ensures that less false positives will be considered by the remaining parts of the framework.

References

[BCT06] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for DBMS testing. *IEEE Trans. Knowl. Data Eng.*, 2006.

[Bi07] C. Binnig, D. Kossmann, E. Lo, and M.T. Özsu,: QAGen. generating query-aware test databases. *SIGMOD*, 2007.

[BKL07] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. *ICDE*, 2007.

[Co01] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson. Introduction to algorithms. *McGraw-Hill Higher Education*, 2001.

[LCM15] H. Li, C. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13), 2015.

[MKZ08] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. *SIGMOD*, 2008.

[PM16] K. Panev and S. Michel. Reverse engineering top-k database queries with PALEO. *EDBT*, 2016.

[Ps15] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri. S4: Top-k spreadsheet-style search for query discovery. *SIGMOD*, 2015.

[Sa10] A.D. Sarma, A.G. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. *ICDT*, 2010.

[Sh14] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. *SIGMOD*, 2014.

[SS94] A. Swami and K.B. Schiefer. On the estimation of join result sizes. *EDBT*, 1994.

[TCP09] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. *SIGMOD*, 2009.

[TCP14] Q. T. Tran, C. Chan, and S. Parthasarathy. Query reverse engineering. *VLDBJ*, 23(5), 2014.

[TP] TPC. TPC benchmarks. http://www.tpc.org/.

[Va01] V.V. Vazirani. Approximation algorithms. *Springer-Verlag New York, Inc.*, 2001.

[Zh13] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. *SIGMOD*, 2013.

Big Data and NoSQL

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 83

Transformations on Graph Databases for Polyglot Persistence with NotaQL

Johannes Schildgen[1] Yannick Krück[2] Stefan Deßloch[3]

Abstract: Polyglot-persistence applications use a combination of many different data stores. Often, one of them is a graph database to model relationships between data items. The data-transformation language NotaQL can be used to define transformations from one NoSQL database to a different one. In this paper, we present a language extension for NotaQL to allow graph transformations, graph analysis, and data migrations on graph databases. NotaQL is schema-flexible, it offers filters and aggregation functions, and it allows for graph traversal and edge creation. Our graph-transformation platform can be used for iterative graph algorithms and bulk processing.

Keywords: Data Transformation Language, Graph Databases, NoSQL

1 Motivation: Graph Databases

NoSQL databases are typically classified in four categories: key-value stores, wide-column stores, document databases, and graph databases. Every NoSQL database has its own data model, a different support for transactions and distribution, and specific benefits and drawbacks. In many enterprises, different NoSQL databases are used in combination to make the best of all. This approach is called *polyglot persistence* [SF12]. As an example, an in-memory key-value store manages frequently-updated page-visit counters, a document database stores user data, and a graph database keeps track of friendship relationships between the users. Storing everything in a graph database would be possible but slow because in distributed environments graph databases typically don't scale as well as other systems and they have higher access costs. Therefore, our document store manages the user data items, and the graph database stores relationships between them. Data-analytic tasks are executed in specific time intervals to extract information from the graph—e.g., the number of friends for each person—and store it into the document database. Alternatively, data from the document store can be analyzed to introduce new edges in the graph database, e.g. to connect people who frequently communicate with each other.

To simplify the development and support the effecient execution of polyglot-persistence applications [Ge14], not just APIs and frameworks are needed, but also languages and platforms that can transform and move data from one data store into another. At the heart of our vision for this data-store interoperability is a language for data transformations that should support a wide range of database systems with all their specific data-model

[1] Technische Universität Kaiserslautern, schildgen@cs.uni-kl.de
[2] Technische Universität Kaiserslautern, y_krueck11@cs.uni-kl.de
[3] Technische Universität Kaiserslautern, dessloch@cs.uni-kl.de

concepts. Different from many classical approaches, this language should not unify different data models and cover only their commonalities, but be extensible and allow to introduce individual language constructs to fully support all of their data model specifics. Otherwise, the rapidly evolving landscape of NoSQL databases is difficult to support. Our language NotaQL [SD15, SLD16] addresses the above requirements for data transformations between different NoSQL stores. The language is concise, easy to learn, schema-flexible, and data-model independent. The latter fact is realized by allowing specific language constructs for every data model and by using an internal data structure that is a superset of all other supported models. In our previous work, we described how so-called *aggregate-oriented* NoSQL stores (i.e., key-value stores, wide-column stores, document databases) are supported as sources and targets for NotaQL transformations.

In this paper, we significantly extend NotaQL to support graph databases. The main challenges we have to address stem from data-model differences. Aggregate-oriented stores focus on storing independent items that are typically accessed by an ID and searched or transformed one after the other. In contrast, graph databases connect items, provide graph-traversal languages that include powerful access methods, and support graph APIs for easy access of property values and related items. Item relationships are natively supported by graph databases and optimized for fast navigation.

The data model of a graph database is a graph in the mathematical sense. $G = (V, E)$ defines the graph with its vertices and edges. Vertices in a so-called *property graph* are semi-structured—like JSON documents in a document store. A vertex can have a list of property-value pairs, and optional labels. An edge connects two vertices. It has a label, a direction and also a list of properties. Figure 1 shows a simple property graph with two vertices connected by one edge.

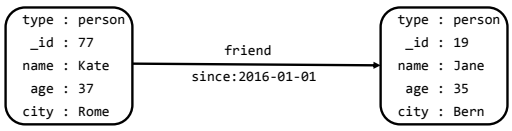


Fig. 1: A simple property graph G_1.

Due to their emphasis on networks of data items, graph databases typically don't scale as well as aggregate-oriented stores. However, they are very important for managing and analyzing complex connected data, which is often found in social networks or recommender systems. Typical operations here are traversing the graph by navigating from one vertex to another and iteratively computing and refining node or edge properties. As a consequence, graph-database languages and platforms differ a lot from classical query languages and computation frameworks. (Please see Section 6 for a more thorough discussion.)

In polyglot-persistence applications that utilize a combination of different systems, frameworks and languages are needed that can handle both aggregate-oriented data models and graphs. Here, one main challenge—in addition to the performance—is how a user can implement algorithms in such a language or framework correctly and efficiently [Ho12]. Lumsdaine et al. [Lu07] state four software design issues, namely flexibility, extensibility, portability, and maintainability. There are many approaches that unify the access to different

stores [Ge14, SGR15, OPV14], but they do not fully support all data-model concepts. Other systems like ArangoDB [Ar16] use a combination of different data models and introduce a query language that supports all of them. In this paper, we present a platform for graph transformations with the language NotaQL. Our platform supports all data-model concepts without introducing a new database system. The NotaQL platform connects to arbitrary NoSQL databases and executes user-defined transformation scripts to perform migrations from one system into another. In graph databases, NotaQL can easily access and modify properties as well as traverse and create edges. Other use cases for NotaQL are data migration and integration tasks. If some database stores relationships as arrays, sub-documents, or foreign keys, and another system uses a graph database, NotaQL can be used to convert the data from the first schema to the second one, and vice versa.

The following list shows the main contributions of our paper:

- We present an easy-to-learn, concise and powerful language for data transformations on property graphs (based on NotaQL),
- a syntax to access and create properties and edges, and to traverse to neighbors,
- a solution for defining iterative graph algorithms,
- a transformations platform to execute NotaQL scripts on different graph database systems,
- an approach for cross-system transformations between graph databases and other kinds of databases or file formats.

In the next section, we provide a brief overview of NotaQL based on previous work. We then present our new extension for graph databases in Section 3. In Section 4, we provide details on implementing graph database support in our transformation platform. After reporting on an initial performance validation in Section 5, we present related work on existing graph languages and frameworks in Section 6 and conclude the paper in Section 7.

2 The Data-Transformation Language NotaQL

NotaQL is a language to transform a set of input items into a set of output items. These sets can be tables in a wide-column store, collections in a document database, or a map in a key-value store. In these cases, the items are rows, documents, or key-value pairs.

The main part of a NotaQL script are a *filter specification* that defines which items to transform, and *attribute mappings*. A NotaQL script is output oriented. This means that the attribute mappings define how the output items should look like, i.e. which attributes they have and how the values of these attributes are computed. Figure 2 shows how a NotaQL script is logically executed. First, the items are filtered by the predicate given in the filter specification in the NotaQL script. Afterwards, intermediate items are created by mapping each item as defined in the attribute mappings. The mapped items are then decomposed into output fragments, and fragments which belong to the same output *cell* are combined with reference to the aggregation functions used in the transformation script.

In a wide-column store, the input items are the rows of a table. The ones that fulfill the filter are split into cells. A cell $z = (\_r, \_c, \_v)$ is defined by the row-id $\_r$ of the row where it

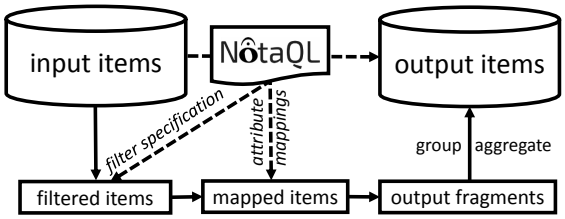


Fig. 2: Logical Execution of a NotaQL Script

originates, a column name $\_c$, and a value $\_v$. The attribute mappings define how to create output cells based on the input cells. As a first example, the following script counts all people older than 17 in each city:

```
IN-FILTER: IN.age > 17,
OUT._r <- IN.city,
OUT.numPeople <- COUNT()
```

The NotaQL script uses a filter to process only rows that have a column called `age` and a value greater than 17 in this column. The filtered rows are split into their cells, and here, only the cell with the column name `city` is of interest. Each city cell is used to produce an intermediate cell $z' = (\_r', \_c', \_v')$ where $\_r'$ is the value of the city cell in the input, $\_c'$ is the string literal `numPeople`, and $\_v'$ in each output fragment a 1 because the aggregation function `COUNT()` is defined as `SUM(1)`. All fragments that belong to the same row (i.e., the same city) are grouped together and their values are summed up to the final value.

The NotaQL platform presented in [SLD16] extends the possibilities of NotaQL by not only supporting wide-column stores, but any kind of aggregate-oriented database. The platform reads from one database system, filters, transforms and aggregates data, and writes its result to another system. The output system can be a different kind of NoSQL database, or it can be the same data store. In the latter case, the results are written into a different collection or table, or it can be the same one as the input to modify data in place. A cross-system NotaQL script starts with the definition of an *input* and an *output engine*. Each engine has a name and specific parameters. As an example, the CSV engine takes the path to a CSV file in a local or distributed file system. After the engine definition, an optional *input-filter* clause can be used to make a selection. The rest of the NotaQL script are *attribute mappings*. They define output attributes and how their values are computed based on the input data. Usually, the first attribute mapping defines the value of an object ID. In key-value stores, the ID is called a *key*; in wide-column stores, it is the *row-id*. They have to be set in a NotaQL script. If an ID value is already present in the database, an in-place update of the existing item is performed, otherwise a new item with the given ID is inserted. Document stores and other system can automatically generate IDs, so there is no need to define an attribute mapping for them in that case. Figure 3 shows the syntax of a cross-system NotaQL script.

notaql:

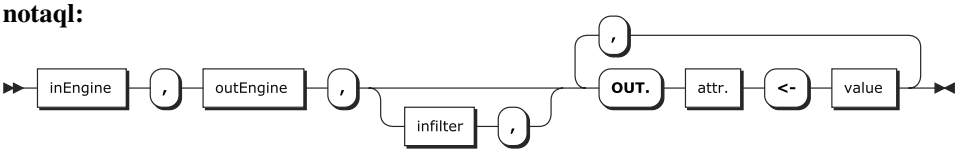


Fig. 3: NotaQL Syntax

The grammar symbols *attr.* and *value* depend on the input and output engine. When—like in the example above—a wide-column store is used as the input, the value can be `IN._r` (row-id), `IN._c` (column name), `IN._v` (column value), or `IN.x` (value of a given column `x`). Analogue for the output attributes. Key-value stores do not have named columns but only keys and values which can be accessed with `IN._k` and `IN._v`. Document stores use `IN._id` for a document ID, dot notation (`IN.x.y`) for accessing sub-attributes, and a function `LIST` to create arrays.

The following example reads all key-value pairs of a Redis database that represent user items and stores them into a MongoDB collection:

```
IN-ENGINE: redis(database <- 0),
OUT-ENGINE: mongodb(database <- 'test', collection <- 'users'),
IN-FILTER: _k LIKE 'user/%',
OUT.username <- IN._k,
OUT.email <- IN._v
```

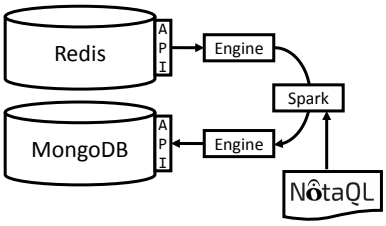


Fig. 4: Cross-System Execution of a NotaQL Script

The access to the key and value using `IN._k` and `IN._v` is specific for key-value databases. For the MongoDB output documents, arbitrary attributes can be set, here `username` and `email`. As no `OUT._id` is set, each document will have an automatically generated document identifier. NotaQL is a schema-flexible language. One can access the values of all attributes without knowing their names using `IN.*`. An attribute name can be returned with `IN.*.name()`. Accessing only specific attributes is possible with *attribute filters*: `IN.?(name()!='age')` finds all attributes except the age. On the output side, the *indirection operator* `$` can be used to create new attributes using a derived attribute name, e.g., `OUT.$(IN.*.name())` creates the same attributes in output items as the ones found in the input items. While iterating over a set of attributes using the `*` or `?`, the attribute of the current iteration can be accessed with `IN.@`. This way, all attributes together with their values can be copied using `OUT.$(IN.*.name()) <- IN.@`.

Figure 4 shows the execution of a NotaQL script on our Apache Spark-based [Za10] platform. The internal data model of NotaQL is based on the JSON data model because it is the most powerful data model of the aggregate-oriented stores. All other supported models and also CSV or JSON files can be mapped to this model. But it comes to its limits when we want to support graph databases. That is why we present a new approach in this paper that extends NotaQL by supporting relationships between items, and performing iterative transformations.

3 NotaQL for Graph Databases

A graph database can be seen as a document store with documents being the vertices and connections between documents being the edges. If we leave out the edges, the data model

is the same as for document databases. So, we can reuse the existing NotaQL language for vertices and properties.

3.1 Item-to-Item (Vertex-to-Vertex) Transformation

We first focus solely on how vertex information can be access and mapped. So edge connections are ignored here and will be covered later in Section 3.2. A NotaQL script defines how to create an output item based on the input. Given a property graph without edges $G_1 = (V, \emptyset)$ stored in a graph database, NotaQL connects to the input graph G_1 and writes its output to an initially empty graph G_2. As described in Section 2, an `IN-FILTER` clause can be used to filter input items by a given predicate. All vertices that fulfill the predicate are transformed with respect to *attribute mappings*. Every attribute mapping has the form `OUT.p <- value`, where `p` is the name of a property of the output vertices and `value` an arbitrary definition or computation of a numeric, string or differently typed value. Here, properties of input vertices can be accessed, literals can be used, and functions can be called.

The following NotaQL script transforms the graph G_1 (the one shown in Figure 1) that is stored in a Neo4J [Ne16a] database into G_2 by performing some selections and projections:

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G2'),
IN-FILTER: type='person' && age > 0,
OUT.name <- IN.name,
OUT.year_of_birth <- 2016 - IN.age,
OUT.type <- 'person'
```

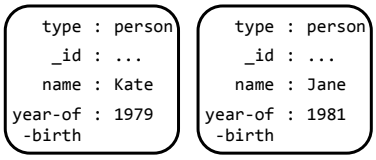


Fig. 5: Selection and Projection of Vertices; Graph G_2

The execution of this transformation works as follows: For every vertex having the label `'person'` and an age greater than zero, an output vertex is created with the same name and a new property `year_of_birth`. All other properties are discarded. As this transformation writes its output to an initially empty graph, the concept of a vertex ID is not needed here. In Neo4J, vertices do have identifiers, so in this case, they are automatically generated. When a NotaQL transformation is used to update an existing graph in place, an ID has to be set to specify whether to change an existing vertex (the one with the given ID) or to insert a new vertex (if no ID matches).

In the following example transformation, the age value in every person vertex with an age greater than zero is incremented by one.

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G1'),
IN-FILTER: type='person' && age > 0,
OUT._id <- IN._id,
OUT.age <- IN.age+1
```

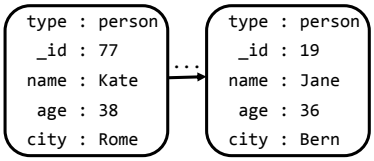


Fig. 6: In-Place Updates on a Graph

The `OUT._id <- IN._id` mapping indicates that the current vertex should be updated. In case the script might produce multiple output vertices with the same ID, the property

mappings have to either guarantee the equality of all values within each ID group, or they have to contain aggregate functions. The aggregation function `SUM`, `COUNT`, `MIN`, `MAX`, and `AVG` are used to generate atomic output values for every property. All property values of output vertices that have the same ID are reduced to a final value using the given function.

The following transformation produces a new graph G_3 that consists of city vertices having the average age of all people living in this city as a property:

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G3'),
IN-FILTER: type='person' && age > 0,
OUT._id <- IN.city,
OUT.city <- IN.city
OUT.avg_age <- AVG(IN.age),
OUT.type <- 'city'
```

```
  type : city
   _id : Rome
  city : Rome
avg_age : 37
```

```
  type : city
   _id : Bern
  city : Bern
avg_age : 35
```

Fig. 7: Average Age per City; Graph G_3

Because of the trivial functional dependency `IN.city` → `IN.city` and the given property mapping, the functional dependency `OUT._id` → `OUT.city` holds. This is why the `city` property values are equal within each ID group. For the property `IN.age`, the aggregation function `AVG` is used to calculate an atomic output value for the output property `OUT.avg_age`.

3.2 Traversing Edges in the Input Graph

As shown above, `IN.` is used to access property values of input vertices. We added the following three steps to access its edges: `IN._e` traverses all edges, `IN._>e` all outgoing, and `IN._<e` all incoming ones. With ?-predicates, which were already used in non-graph NotaQL (see Section 2), the kinds of edges can be further specified; based on their labels and properties. After an edge step, one can access the edge properties with the simple dot notation, e.g. `IN._e.since`, and one can follow the edge to its neighbor vertex using one more _ symbol. For selecting only specific neighbors, additional ?-predicates can be used. The neighbor vertex's properties and edges can be accessed as usual. E.g., `IN._e_.name` returns the neighbor's name. When iterating over multiple edges, the edge within the current iteration can be accessed using `IN._e[@]`. To avoid long lists of edge steps, NotaQL uses the `[min,max]` option to follow an edge over multiple hops. The default edge traversal is exactly one hop: `[1,1]`. For the transitive closure up to an unbounded number of hops, `[0,*]` is used. In that case, termination in cyclic graphs has to be achieved by using proper predicates. Figure 8 shows the grammar for edge traversals in NotaQL.

Aggregation functions in NotaQL take a list of values and compute one output value. In the examples in Section 3.1, the lists where created due to the cell groupings. All values that belong to the same output cell—i.e., in a graph database, the same property for the same output vertex—, are collected in a list, and an aggregation function combines the values in this list. NotaQL also allows calling aggregation functions on lists that are not created after grouping but already existed in the input data. These lists are automatically created when using ambiguous expressions, e.g. `IN.a[*]` contains all elements of an array, `IN._e_._id` are the IDs of all neighbor vertices.

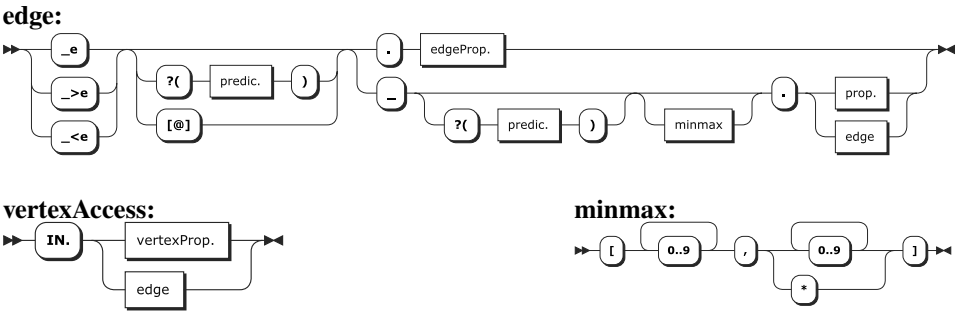


Fig. 8: Syntax for Traversing Edges and Access to Properties

The following NotaQL script shows an example with some edge accesses and aggregations.

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G1'),
IN-FILTER: type='person',
OUT._id <- IN._id,
OUT.num_neighbors <- COUNT(IN._e_._id),
OUT.num_old_friends <- COUNT(IN._e?('friend')_?(age>=80)._id),
OUT.num_friends_of_friends <- COUNT(IN._e?('friend')_[1,2]._id),
OUT.oldest_friendship_date <- MIN(IN._e?('friend').since),
OUT.mother_name <- IN._>e?('mother')_.name
```

The edge predicate `?('friend')` is a short form for `?(_l = 'friend')`, where `_l` designates the edge label. Most edge traversals in this example use the `_e` path to traverse an edge independently of its direction. However, to navigate to the mother's vertex, the `_>e` path selects only outgoing edges. The aggregation functions used in this example are `COUNT` and `MIN`. They are called on lists of property values of neighbor vertices, respectively on edge properties. When a NotaQL transformation groups multiple vertices, it is possible that values are lists before grouping. This results in lists of lists after grouping. These are resolved by using two aggregation functions in combination. The following example shows how to compute the average number of friends people have per city:

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G3'),
OUT._id <- IN.city,
OUT.avg_num_friends <- AVG(COUNT(IN._e?('friend')_._id))
```

The expression `IN._e?('friend')_._id` is a list of friend IDs. After grouping by city, every city vertex contains a list of lists of these IDs. The cardinalities of the inner lists are computed with the `COUNT` function. The `AVG` function computes the average of these count values.

3.3 Creating Edges

For traversing edges or accessing their properties, we handle edges almost like vertex properties, as seen above. However, creating an edge is not as trivial as setting a vertex

property. This is because an edge has a target vertex, a label, a direction, and a list of properties. We decided to define the target and direction of an edge on the left-hand side of the mapping arrow `<-`, and the edge data as parameters of an `EDGE` constructor on the right-hand side. This approach is consistent with NotaQL's support for other data models and data types, where the construction of complex objects or list values is specified in a similar way. The target of an edge is determined by a predicate that matches the target vertex. If multiple vertices match the predicate, then multiple edges are created. Figure 9 shows the syntax for edge creation.

outEdge:

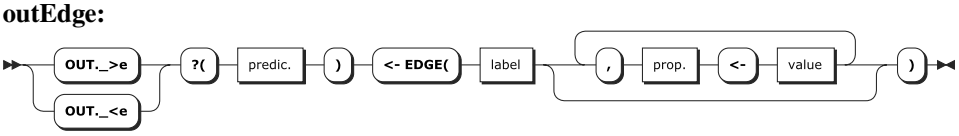


Fig. 9: Syntax for Edge Creation

From the current vertex's view, `OUT._>e` creates an outgoing and `OUT._<e` an incoming edge to respectively from the vertices that fulfill the given predicate. The `EDGE` constructor function requires a label and a (possibly empty) list of edge properties.

In the following example, two edges are created for every vertex: The first one says that everybody is a friend of Sam, and the second one is an edge to each person's grandmother[4]:

```
OUT._id <- IN._id,
OUT._>e?(name = 'Sam') <- EDGE('friend'),
OUT._>e?(_id = IN._>e?('mother' || 'father')_._>e?('mother')_._id)
   <- EDGE('grandmother', via <- IN._>e[@]._l)
```

Within the latter ?-predicate, the ID of the grandmother vertices are searched, and edges are created to the vertices having this ID. The edge is an outgoing one; it has the label `grandmother` and one property `via`. The term `IN._e[@]._l` navigates to the current edge that is bound in the edge-target predicate and reads its label. So the value of the `via` property is either `'mother'` or `'father'`. If a vertex does not have a mother or father edge, or if the neighbor does not have a mother edge, the predicate is evaluated to false for every vertex. In this case, no grandmother edge is created. Multiple paths to the same vertex do not result in the creation of multiple edges. But if a person has multiple mother or father edges to different vertices, the equality predicate checks for list containment so that multiple edges would be created. Same if more than one person is named Sam. Then friendship edges are created to all of them.

3.4 Iterative Computations

Many graph algorithms that change the graph in place have to be executed multiple times to produce the final result. In NotaQL, we therefore introduce the `REPEAT: n` clause, which can be used to run a transformation `n` times. The repeat iteration also stops if the graph has not changed since the previous iteration. For `REPEAT: -1`, the computation runs as long as

[4] In this and the following examples, we omit the `IN-ENGINE` and `OUT-ENGINE` clause when the output graph is the same as the input graph, i.e., for in-place updates. We also omit the `IN-FILTER`, which should be used to only apply the transformation on vertices with a specific label.

the graph changes. Another possibility is to monitor the change of a single given vertex property (`property(p%)`), and to stop as soon as the change of this property value is below a certain percentage `p` for all vertices. Figure 10 shows the syntax of a `REPEAT` clause.

repeat:

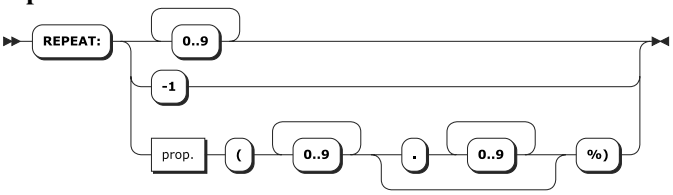


Fig. 10: Repeat Clause

The following example shows the PageRank [Pa99] algorithm in NotaQL. It uses a global function `NumVertices()`, which returns the number of vertices in the graph.

```
OUT._id <- IN._id,              # initialization
OUT.pagerank <- 1/NumVertices();

REPEAT: pagerank(0.0005%),      # main iteration
OUT._id <- IN._>e_._id,
OUT.pagerank <- SUM(IN.pagerank/count(IN._>e._id))
```

$$PR(p) = \frac{1}{N}$$

$$PR(q) = \sum_{p \in in(q)} \frac{PR(p)}{|out(p)|}$$

Fig. 11: PageRank

There are two transformations. After having initialized the graph with `pagerank` $= 1/N$ for all vertices—with N being the number of vertices—the iterative part of this NotaQL script is then executed until all PageRank values change less than 0.0005% within one iteration. For a given input vertex p, the PageRank values of all neighbors $q \in out(p)$ via the outgoing edges are influenced. The final value of q's PageRank is the sum of all PageRank values from vertices like p divided by their out-degree. It can be seen in Figure 11 that the NotaQL definition is very similar to the original PageRank formula. Improvements like a damping factor [Pa99] can easily be added into the NotaQL script.

3.5 Modeling Relationships in Non-Graph Databases

Not all applications use graph databases to store relationships between data items. We want to present three popular schema approaches that are often found in relational and NoSQL databases to model graphs using data-model concepts like tables, lists or nested objects. During this discussion we show (non-graph) NotaQL scripts that work with these schema approaches and transfer one representation into another. Later, we present cross-system transformations between a graph database and a non-graph database that uses one of the three presented schemes.

Vertex and Edge Table This schema is often found in relational databases to model graphs. A vertex table has a primary-key column for a vertex ID, a type column for the label, and one column for each property. The edge table consists of two foreign-key columns for the source and target vertex IDs, also one column for the label, and one for each property. For querying and transforming the data, the two tables need to be accessed multiple times

and joined to traverse the graph, once for every hop. As NoSQL databases try to avoid joins, this approach is mostly used in relational databases. Furthermore, the properties have to be defined in advance, and every vertex can only have one label. This can be solved by introducing separate label and property tables [Su15]. Tables 1 and 2 show the schema of the graph shown in Figure 1.

id	**type**	**name**	**age**	**city**
77	person	Kate	37	Rome
19	person	Jane	35	Bern

Tab. 1: Vertex Table

source	**target**	**label**	**since**
77	19	friend	2016-01-01

Tab. 2: Edge Table

Adjacency Lists Adjacency lists and adjacency matrices are the most common ways to represent graphs for mathematical computations. While adjacency matrices are rarely used in databases or files, the lists are often found to store graphs, e.g., in CSV files. Every line in this file represents a vertex. The first value in one row contains a vertex ID, the other values are IDs of neighbors via outgoing edges:

```
77,19,28,39,21
19,40,28
```

The same graph can be represented in a wide-column store (see Table 3). Here, multiple column families can be used to both store vertex properties and neighbors in one table [Ch08]. Instead of leaving the column values empty, they can also hold edge properties.

row-id	**friends**				**info**		
77	19:-	28:-	39:-	21:-	name:Kate	age:37	city:Rome
19	40:-	28:-			name:Jane	age:35	city:Bern

Tab. 3: Adjacency Lists in a Wide-Column Store

Adjacency lists are also often found in document databases in the form of arrays. Again, vertex properties and outgoing edges can be stored within one single JSON document. We want to show a NotaQL transformation to convert a vertex and edge table into this schema. We do this by taking the edge table (see Table 2) as the input, construct lists of neighbor-vertex ids, and write these lists in the original vertex table (see Table 1).

```
OUT._id <- IN.source,
OUT.$(IN.label) <- LIST(IN.target)
```

The indirection operator $ is used to produce individual lists for every edge label. The result documents look like this:

```
{ _id: 77, name: "Kate", age: 37, city: "Rome",
  friend: [ 19, 28, 39, 21 ] }
```

Nested Objects Forming Tree Structures If the data items form tree structures, relationships between items can be modeled by nesting them. In a document store, we use an array of sub-documents to model edges to these sub-documents. The following example shows a blog post with comments that can again be commented:

```
{ _id: 1420733, user:  "Kate", text: "I'm in Berlin now",
  comments: [{ user: "Carl", text: "Have fun!" },
             { user: "Jim", text: "I'm also in Berlin!",
               comments: [{ user: "Kate", text: "Let's meet!" }]}]}
```

All these variants are often found in databases. But for complex graphs, they are not applicable. This is because writing programs and queries over such schemata is hard, analysis and transformations are slow, and joins are often not supported. The solution are graph databases which offer special query languages and a better performance.

3.6 Cross-System Graph Transformations

In polyglot-persistence environments, it is often necessary to transfer or copy data from one data store to another. As graph databases exhibit worse horizontal scalability than for example document stores, it is a popular approach to use a graph database only for storing relationships between items, while the item's properties are stored in a document store. The grammar in Figure 12 is a modified version of Figure 3 from the beginning of this paper. With the graph-database extension of NotaQL, it is possible to mix the usage of graph and other databases in the `IN-ENGINE` and `OUT-ENGINE` definition. For each engine, data–model-specific language constructs can be used, e.g. `IN._k` and `IN._v` for a key-value store or `IN._e` for a graph database as input. All this is part of the definition of the `value` symbol in Figure 12.

notaql:

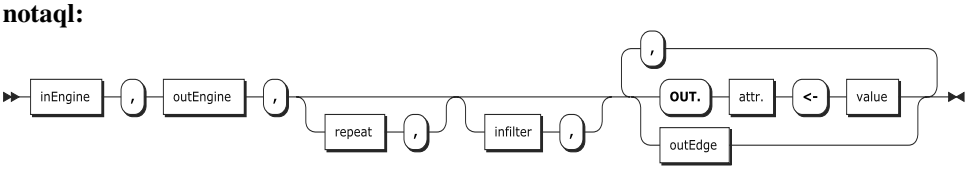


Fig. 12: Syntax for Cross-System NotaQL Transformations with Edge-Creation Option

In the next example, we want to show a typical data transformation, namely a data-migration task from MongoDB to Neo4J. As MongoDB does not support direct relationships between documents, a one-to-n relationship for a person's status updates can be modeled as a list of nested sub-documents, and an n-to-m relationship for friendships can be modeled as an adjacency list of the friends' IDs. Figure 13 shows one input document and the desired output of the documents-to-graph transformation.

Using the following script, the data is transformed into a graph with person and status vertices plus the connecting edges.

```
#1. create status vertices
IN-ENGINE: mongodb(database<-'socialnet', collection<-'people'),
OUT-ENGINE: neo4j(path <- '/data/socialnet'),
OUT._id <- IN.status[*].sid, OUT.ts <- IN.status[@].ts,
OUT.text <- IN.status[@].text, OUT.type <- 'status';
```

```
{ _id: 77, name: "Kate", age: 37,
 status: [
  { sid:714, ts:1448269379,
    text:"I'm in Berlin now!!!"},
  { sid:788, ts:1448372647, text:
    "The currywurst museum is amazing!"}
 ], friends: [ 19, 78, 81, 103 ]
}
```

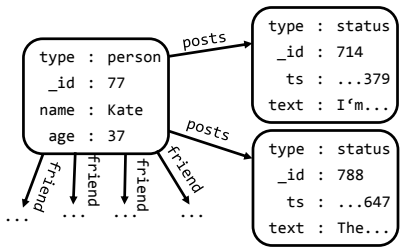


Fig. 13: Left: Input as JSON Documents; Right: Output as a Graph

```
#2. create person vertices and edges to their statuses and friends
IN-ENGINE: mongodb(database<-'socialnet', collection<-'people'),
OUT-ENGINE: neo4j(path <- '/data/socialnet'),
OUT._id <- IN._id, OUT.name <- IN.name, OUT.age <- IN.age,
OUT._>e?(type='status' && _id=IN.status[*].sid) <- EDGE('posts')
OUT._>e?(type='person' && _id=IN.friends[*]) <- EDGE('friend'),
OUT.type <- 'person';
```

There are two transformations. The first one creates status vertices. A new vertex is created for every element in each person's status list. With `[*]`, we iterate over the list. While iterating, the current element can be accessed with `[@]`. For the example document in Figure 13, two status vertices are created. In the second transformation, the person vertex is created. There, an edge links to every status vertex and other edges to the friends. As we explain in the next section, edges are created at the very end of a transformation. So, it is guaranteed in this case, that the target vertices of friendship relationships exist—assumed that there are no dangling references in the input database. Usually, the list of friends in the input is symmetric, so Kate's friend Jane will have Kate's ID in its friends list. To avoid the creation of edges in both directions, one can simply add `&& _id>IN._id`. Then, the friendship edge points from the vertex with the smaller ID to the one with the larger one. The typical way to model symmetric relationships in graph databases is creating an edge in an arbitrary direction. At query time, an edge is traversed independently of its direction.

A transformation in the opposite direction, i.e. from a graph database to a different database, is possible with the edge-accessing steps shown in Section 3.2. To reverse the transformation in Figure 13, the following NotaQL script can be used:

```
IN-ENGINE: neo4j(path <- '/data/socialnet'),
OUT-ENGINE: mongodb(database<-'socialnet', collection<-'people'),
IN-FILTER: type='person',
OUT._id <- IN._id,
OUT.name <- IN.name, OUT.age <- IN.age,
OUT.status <- LIST(OBJECT(sid <- IN._>e?('posts')_._id,
 ts <- IN._e[@]_.ts, text <- IN._e[@]_.text)),
OUT.friends <- LIST(IN._e?('friend')_._id)
```

The transformation is performed for every person vertex. It creates person documents in which the `status` attribute is a list of objects, filled with property values of neighbor vertices using the outgoing `posts` edges. For friends, only the vertex IDs are retrieved to create a list of foreign keys. In the given example, we used the constructor functions `LIST` and `OBJECT`, which are specific for document stores as an output.

As there are also input and output engines for CSV and JSON files, NotaQL can be used to easily import and export graphs.

4 Realization

While our implementation in [SLD16] is based on Apache Spark [Za10], we developed our graph transformation prototype as a simple Java tool. This is because we found no good framework that supports the different data models of graphs and simple datasets properly. The other reason is that graph databases are usually accessed with their own API and not with frameworks because they are typically not stored in a distributed system with multiple machines. We decided not to use the direct API of a graph database but to use the graph-database gateway *Tinkerpop Blueprints* [Ap16] (see Figure 14). This gateway gives a unified access to different graph database systems like Neo4J, Titan [Ti16], Tinkergraph [Ap16] and many others. This way, we support a magnitude of various stores at the same time. One drawback is that Blueprints does not support vertex labels. As seen in the previous examples, we used a `type` property for storing it.

A graph-to-graph transformation is done in five steps:

1. Parsing
2. Vertex Access and Pre-Filter
3. Input Filter
4. Creation of Vertices
5. Creation of Edges

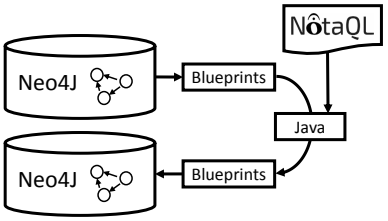


Fig. 14: NotaQL-Script Execution on Graphs

In iterative algorithms, the steps (2) to (4) are repeated until the stop condition is fulfilled. For parsing in step (1), we use the ANTLR [Pa16] parser and lexer. Step (2) uses the Blueprints method `getVertices(String key, Object value)` to find a superset of all vertices of interest. As this method only checks for equality on one single property, complex input-filter predicates have to be converted into the conjunctive normal form (CNF) first. After that, one clause that only consists of one equality literal, can be used for pre-filtering with the `getVertices` method. If there is no such single-equality-literal clause, or if there is no input filter at all, all vertices are received. But in fact, in typical NotaQL transformations there is such a equality literal, namely a type predicate (e.g., `type = 'person'`).

In step (3), our algorithm iterates over all found vertices from step (2) and filters out the ones that violate the input filter. If Blueprints adds a support for complex predicates in the future, or if we change the implementation so that it uses the native graph-database API, step (3) can be dropped.

Within the iteration over the input vertices, new vertices are created and the attribute mappings in the NotaQL script are performed (step (4)). Simple property mappings like `OUT.a <- IN.b` or `OUT.a <- 5` can be easily evaluated to create an output vertex's property values. Traversals over edges in the input graph are converted into method calls of the Blueprints API to navigate to neighbors. Aggregation functions that are applied on lists in the input can also be easily evaluated by retrieving all values from neighbors and computing the aggregated result, e.g. a sum. Aggregation functions after grouping are more complex to evaluate as they combine data from multiple input vertices. One solution would be grouping all output vertices by their ID and afterwards applying a reduce function to produce one single vertex for every group. This behavior is similar to MapReduce [DG04] or Spark [Za10]. But these frameworks split the data into chunks and distribute them over a cluster of machines. In our case, all data has to be kept in memory of our computing node. Our solution is simple and effective: The aggregation functions are ignored first. When the output vertex is written into the target graph database, this write is done in an incremental fashion with reference to the given aggregation function. For example, if the function is `MIN`, the property value is overwritten only if it is smaller than the current one. If it is `SUM`, the current property value is incremented by the new one.

Step (5) starts as soon as all vertex creations are completed. In that step, our algorithm iterates again over all input vertices that fulfill the input filter. Then it evaluates the `OUT._id` definition and all edge creation parts. With this information, it can produce all edges between vertices in the output graph.

For cross-system transformations, we combine our existing Spark-based platform [SLD16] and the graph-to-graph transformation platform presented in this paper. This way, we combine the benefits of both worlds: The first system is optimized for aggregated stores and files and supports distributed storage and computation. The second one works directly on graphs so that connections between data items are supported as native data-model concepts. Both platforms communicate with each other using a common in-memory graph database Tinkergraph [Ap16] as an intermediate format. A transformation from a graph to a non-graph system uses the given NotaQL script to produce a Tinkergraph without edges. After that, a generic Tinkergraph-to-JSON tool is executed to produce a JSON file that can be read by the Spark-based platform to write the output in the target database. A transformation in the opposite direction, i.e., from a non-graph to a graph database, produces the intermediate JSON file first. This one is converted into a Tinkergraph, and the Tinkergraph is the input for a graph-to-graph transformation that produces the output. Both the intermediate JSON file and Tinkergraph do not contain edges. We perform query rewriting to move the edge access and creation

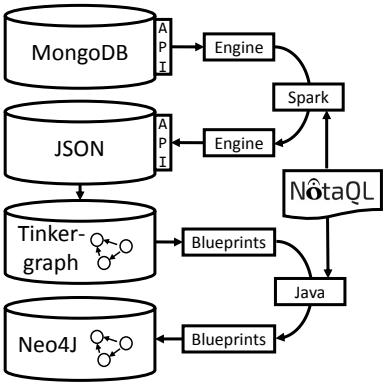


Fig. 15: Cross-System Execution of NotaQL between a Graph and a different NoSQL Database.

part of a NotaQL script to the graph platform. The result is a fast execution that involves different kinds of databases and data models.

Figure 15 shows the execution of a transformation from a MongoDB collection to the graph database Neo4J. The NotaQL script is rewritten so that it writes the (filtered) MongoDB documents into a JSON file. After that, the generic JSON-to-Tinkergraph tool is used to create a Tinkergraph. This Tinkergraph acts as an input for the NotaQL graph-transformation platform that writes the result into Neo4J using the rewritten NotaQL script.

5 Initial Validation

We ran some initial experiments to validate our general approach in terms of feasibility and performance, For that, we used a single machine with a double-core Intel i3 M 370, 2.4 GHz and 3 GB of memory. As a test database, we used a subset of the data of the Slovenian social network Pokec [TZ12] in JSON format. Our test dataset consists of 15,000 vertices and 200,000 edges. Different from our examples above, friendships are asymmetric, so if A has a friend B, this does not imply that B also has a friend A. The data from [TZ12] consists of one collection of person documents, and one containing the relationships between them. This corresponds to the vertex/edge-table schema described in Section 3.5. In a first step, we combined both collections into person documents with adjacency lists. These documents look like this:

```
{ "_id":"14943", "user-id":"14943", "username":"14943",
  "avatar":"185 cm, 65 kg, ????asi:d",
  "follower": ["8303"], "following": ["8934","544"] }
```

In [Pa15], a complex Java program is presented to import such a graph that is stored in a JSON file into Neo4J. The program iterates over all JSON documents and creates Cypher statements to create vertices for every person and statements to add the edges between them. The list of statements is written into a file, which is then executed on Neo4J. We tested the proposed program on our Pokec database. It took 207 minutes to import the data into Neo4J. With NotaQL, the transformation can be expressed in only four short lines of code and the import needed only 95 seconds. In applications where we have much larger data sets, the solution that generates Cypher statements is not suitable because it would need many days for the import.

In a second experiment, we imported the Pokec dataset into a MongoDB database. We used the MongoDB aggregation pipeline to compute the number of friends plus the number of friends of friends for every person. The algorithm starts with unwinding the `following` array, grouping by the ID, and counting the direct neighbors. Then, the number of indirect neighbors are added to this number by performing a `$lookup` operation and another grouping and counting. This computation took 23 minutes. Next, we formulated the query as an NotaQL script. As neither the input nor the output is a graph database, we cannot apply the NotaQL graph-transformation platform here directly. So, we first load the MongoDB collections into an in-memory Tinkergraph, and apply the friends-of-friends computation

on this, using the MongoDB output engine. All in all, this job took only 40 seconds, so we have a speed-up of more than 3,000%.

6 Related Work: Graph Languages and Frameworks

The three most common ways to work with a graph database are (1) working with the database-specific API to find, create, and modify vertices and edges of a graph, (2) using a graph query language, and (3) utilizing a graph-processing framework. The API methods provide efficient access to individual vertices and support a simple graph traversal. Application developers can directly use these methods to work with graph databases. However, for complex tasks like data analytics or iterative computations, much code needs to be written. A NotaQL script can be written in one minute and executes the same complex graph transformation. As an alternative to APIs, graph query languages like Gremlin and Cypher can be used. *Gremlin* [Ro15] belongs to the *Tinkerpop* [Ap16] stack and uses the common *Blueprints* API for graph databases. Thus, Gremlin is widely supported. A query consists of multiple steps such as filters to select vertices and edges based on their labels or properties, traversal steps to move to neighbor edges or vertices, or aggregation steps. Side-effect steps can be used to store intermediate values in variables so that they can be accessed in later steps. Gremlin is easy-to-use and powerful for reading, but not for graph transformations. For vertex and edge creation, there are only simple methods that are typically only used to modify one single vertex or edge. NotaQL also uses the Blueprints API and thus, it supports all graph databases that are supported by Gremlin. However, NotaQL is used for the tasks Gremlin cannot do, namely complex graph transformations. Another language is *Cypher*, the graph query language for *Neo4J* [Ne16a]. Its syntax is similar to SQL and consists of pattern-matching elements like in *SPARQL* [Pr08]. A `MATCH` clause is used to define a pattern that is searched for in the whole graph. In this pattern, variable names can be introduced. For every match, the rest of the query is executed. This can be a `WHERE` clause for filtering, a `RETURN` clause to return a result for each match, or a writing clause like `UPDATE`, `CREATE` or `DELETE` to modify the graph in place. While this allows for set-oriented graph transformations, Cypher has restrictions with respect to the complexity of these transformations. In writing clauses, no aggregation functions can be used. Furthermore, Cypher does not support iterative algorithms, it cannot properly handle flexible schemata or transform metadata into data and vice versa. NotaQL supports all these features and it allows for cross-system transformations. Cypher queries can only be based on one single graph, and they can only change the graph in place.

There are many graph-processing frameworks that are optimized for distributed graph processing. *Pregel* [Ma10] and its open-source implementation *Apache Giraph* [Av11] are frameworks to define an iterative algorithm as a user-defined function that is called on every vertex. Within this function, the properties and neighbors of a vertex can be accessed, properties can be modified, messages can be sent to neighbors, and incoming messages from the previous iteration can be processed. These frameworks are typically not used for computations on graph databases but on graphs stored in a file, a relational database or a wide-column store. NotaQL supports both graphs in one of these formats and also graph databases. Furthermore, there is no need to have programming skills. One simple writes a

concise script to map input data to the output. The data-processing framework *Apache Spark* [Za10] has a component called *GraphX* [Xi13] which supports iterative graph analysis similar to Pregel. The graph has to be stored in a vertex table and an edge table. However, again, this framework is not used to work on graph databases. There are Spark connectors for graph databases, e.g. for Neo4J, but they offer a flattened look on the graph without the information about vertex neighbors. Edges can be traversed by writing Cypher queries within Spark methods. In our graph extension for NotaQL, edges are first-class citizens. It natively supports the full graph data model and not just a greatest common divisor between graphs and aggregate stores.

Spark can be used to implement a cross-system transformation from a graph database to a different database. However, these kinds of transformations do not work in the opposite direction, and they only work on simple graphs. There are special tools for graph imports and exports, but no generic ones like our NotaQL platform. The Neo4J Doc Manager [Ne16b] is a tool that loads documents from MongoDB [Mo16] into the graph database Neo4J. A vertex is created for every document d and one for every of their sub-documents s with an edge between d and s. This is very restricted and requires an extra effort on the source and target side before and after the import process to bring the data in the desired format. With the other languages and frameworks presented above, graph transformations to or from a different kind of data store are not possible.

Green-Marl [Ho12] is a language for graph analysis. It can be used to compute scalar values from the graph or a property for every vertex. Users can develop algorithms in a few lines of code which is then optimized and compiled into efficient parallel C++ code. Green-Marl is not a query or transformation language, but a programming language especially for graphs. Thus, users need to write code that defines *how* to compute a result, not *what* the result should be—as in SQL or NotaQL. In the paper, the authors claim that the PageRank algorithm can be expressed with 15 lines of code, in contrast to its native implementation in C++, which has 58 lines. As shown in Figure 11, the PageRank definition in NotaQL has only 5 lines of code.

7 Conclusions and Future Work

As a summary, we presented a language extension for NotaQL to define graph transformations as short transformation scripts. This way, we showed the extendability of NotaQL for complex data models. The language is more powerful than existing graph languages, it supports graph traversal, edge access and creation, and iterative algorithms. NotaQL can be used to modify a graph in place, to produce new graphs, and perform graph analytics using groupings and aggregations. We presented an approach for cross-system graph transformations between a graph database and any kind of NoSQL database or file format. For this, we used system-specific NotaQL engines and a combination of the Apache Spark framework and the Blueprints API for graphs. Our data–model-independant language fully supports the different data models, not the greatest common divisor of those. For that, we use tailored access paths for each model: graphs, documents, key-value pairs, tables, files, and more.

The most complex ones are graphs. For those, we presented an powerful and intuitive syntax to work with edges, neighbor vertices and their properties.

Our results show that our platform is very fast in graph transformations. This is because a NotaQL script is directly executed on a graph database using an API, not an intermediate query language. Cross-system transformations between graph and non-graph databases use the combination of the Blueprints API and the Apache Spark framework for distributed computations.

The NotaQL graph-transformation platform fulfills the four requirements for graph-processing software described in [Lu07]. It is *flexible* regarding data models and the data schema, *extensible* through user-defined engines and functions, *portable* to different database systems, and *maintainable* as a NotaQL script is concise and well understandable.

For future work, we want to improve the performance by integrating our Spark-based and graph-transformation platform closer. Instead of using an intermediate JSON file, we want to build a Tinkergraph engine for the Spark-based platform. This way, the in-memory graph database Tinkergraph can be accessed with Spark like a document store. After that, we plan more performance evaluations and tests. As the NotaQL language is independent of its implementation, more efficient implementations based on systems like Apache Giraph [Av11] or Green-Marl [Ho12] can follow.

References

[Ap16] Apache Tinkerpop: 2016. `http://tinkerpop.apache.org`.

[Ar16] ArangoDB: 2016. `https://www.arangodb.com`.

[Av11] Avery, Ching: Giraph: Large-scale graph processing infrastructure on hadoop. Proceedings of the Hadoop Summit. Santa Clara, 11, 2011.

[Ch08] Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson C; Wallach, Deborah A; Burrows, Mike; Chandra, Tushar; Fikes, Andrew; Gruber, Robert E: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2):4, 2008.

[DG04] Dean, Jeffrey; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. OSDI, pp. 137–150, 2004.

[Ge14] Gessert, Felix; Friedrich, Steffen; Wingerath, Wolfram; Schaarschmidt, Michael; Ritter, Norbert: Towards a Scalable and Unified REST API for Cloud Data Stores. In: GI-Jahrestagung. pp. 723–734, 2014.

[Ho12] Hong, Sungpack; Chafi, Hassan; Sedlar, Edic; Olukotun, Kunle: Green-Marl: a DSL for easy and efficient graph analysis. In: ACM SIGARCH Computer Architecture News. volume 40. ACM, pp. 349–362, 2012.

[Lu07] Lumsdaine, Andrew; Gregor, Douglas; Hendrickson, Bruce; Berry, Jonathan: Challenges in parallel graph processing. Parallel Processing Letters, 17(01):5–20, 2007.

[Ma10] Malewicz, Grzegorz; Austern, Matthew H; Bik, Aart JC; Dehnert, James C; Horn, Ilan; Leiser, Naty; Czajkowski, Grzegorz: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, pp. 135–146, 2010.

[Mo16] MongoDB: 2016. https://www.mongodb.org.

[Ne16a] Neo4J: 2016. https://www.neo4j.com.

[Ne16b] Neo4j Doc Manager: 2016. https://neo4j.com/developer/neo4j-doc-manager.

[OPV14] Ong, Kian Win; Papakonstantinou, Yannis; Vernoux, Romain: The SQL++ Query Language: Configurable, Unifying and Semi-structured. arXiv preprint arXiv:1405.3631, 2014.

[Pa99] Page, Lawrence; Brin, Sergey; Motwani, Rajeev; Winograd, Terry: The PageRank Citation Ranking: Bringing Order to the Web. (1999-66), November 1999. Previous number = SIDL-WP-1999-0120.

[Pa15] Paksoy, Volkan: From JSON to Neo4J. 2015. http://volkanpaksoy.com/archive/2015/09/18/from-json-to-neo4j.

[Pa16] Parr, Terence: ANTLR. 2016. http://www.antlr.org.

[Pr08] Prud'Hommeaux, Eric; Seaborne, Andy et al.: SPARQL query language for RDF. W3C recommendation, 15, 2008.

[Ro15] Rodriguez, Marko A: The Gremlin graph traversal machine and language. In: Proceedings of the 15th Symposium on Database Programming Languages. ACM, pp. 1–10, 2015.

[SD15] Schildgen, Johannes; Deßloch, Stefan: NotaQL Is Not a Query Language! It's for Data Transformation on Wide-Column Stores. In: British International Conference on Databases – BICOD 2015. 7 2015.

[SF12] Sadalage, Pramod J.; Fowler, Martin: NoSQL Distilled: A brief guide to the emerging world of polyglot persistence. Addison-Wesley Professional, 1st edition, 2012.

[SGR15] Schaarschmidt, Michael; Gessert, Felix; Ritter, Norbert: Towards Automated Polyglot Persistence. Datenbanksysteme für Business, Technologie und Web (BTW), 2015.

[SLD16] Schildgen, Johannes; Lottermann, Thomas; Deßloch, Stefan: Cross-system NoSQL data transformations with NotaQL. In: Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. ACM, p. 5, 2016.

[Su15] Sun, Wen; Fokoue, Achille; Srinivas, Kavitha; Kementsietsidis, Anastasios; Hu, Gang; Xie, Guotong: SQLGraph: an efficient relational-based property graph store. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 1887–1901, 2015.

[Ti16] Titan: Distributed Graph Database. 2016. http://titan.thinkaurelius.com.

[TZ12] Takac, Lubos; Zabovsky, Michal: Data analysis in public social networks. In: International Scientific Conference and International Workshop Present Day Trends of Innovations. pp. 1–6, 2012.

[Xi13] Xin, Reynold S; Gonzalez, Joseph E; Franklin, Michael J; Stoica, Ion: Graphx: A resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems. ACM, p. 2, 2013.

[Za10] Zaharia, Matei; Chowdhury, Mosharaf; Franklin, Michael J; Shenker, Scott; Stoica, Ion: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. volume 10, p. 10, 2010.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 103

Distributed Grouping of Property Graphs with Gradoop

Martin Junghanns[1] André Petermann[2] Erhard Rahm[3]

Abstract: Property graphs are an intuitive way to model, analyze and visualize complex relationships among heterogeneous data objects, for example, as they occur in social, biological and information networks. These graphs typically contain thousands or millions of vertices and edges and their entire representation can easily overwhelm an analyst. One way to reduce complexity is the grouping of vertices and edges to summary graphs. In this paper, we present an algorithm for graph grouping with support for attribute aggregation and structural summarization by user-defined vertex and edge properties. The algorithm is part of Gradoop, an open-source system for graph analytics. Gradoop is implemented on top of Apache Flink, a state-of-the-art distributed dataflow framework, and thus allows us to scale graph analytical programs across multiple machines. Our evaluation demonstrates the scalability of the algorithm on real-world and synthetic social network data.

Keywords: Graph Analytics, Graph Algorithms, Distributed Computing, Dataflow systems

1 Introduction

Graphs are a simple, yet powerful data structure to model and to analyze relationships among real-world data objects. The flexibility of graph data models and the variety of existing graph algorithms made graph analytics attractive to different domains, e.g., to analyze the world wide web or social networks but also for business intelligence and the life sciences [Ne10, Ma10, Pa11, Pe14a]. In a graph, entities like web sites, users, products or proteins can be modeled as vertices while their connections are represented by edges.

Real-world graphs are often heterogeneous in terms of the objects they represent. For example, vertices of a social network may represent users and forums while edges may express friendships or memberships. Further on, vertices and edges may have associated properties to describe the respective object, e.g., a user's age or the date a user became member of a forum. Property graphs [RN10, An12] are an established approach to express this kind of heterogeneity. Figure 1(a) shows a property graph that represents a simple social network containing multiple types of vertices (e.g., *User* and *Forum*) and edges (e.g., *follows* and *memberOf*). Vertices as well as edges are further described by properties in the form of key-value pairs (e.g., *name : Alice* or *since : 2015*). However, while small graphs are an intuitive way to visualize connected information, with vertex and edge numbers increasing up to millions and billions, it becomes almost impossible to understand the encoded information by mere visual inspection. Therefore, it is essential to provide graph grouping methods that reduce complexity and support analysts in extracting and understanding the underlying information [THP08, Ch08, Zh11]. The following examples highlight the analytical value of such methods:

[1] University of Leipzig, Database Group & ScaDS Dresden/Leipzig, junghanns@informatik.uni-leipzig.de
[2] University of Leipzig, Database Group & ScaDS Dresden/Leipzig, petermann@informatik.uni-leipzig.de
[3] University of Leipzig, Database Group & ScaDS Dresden/Leipzig, rahm@informatik.uni-leipzig.de

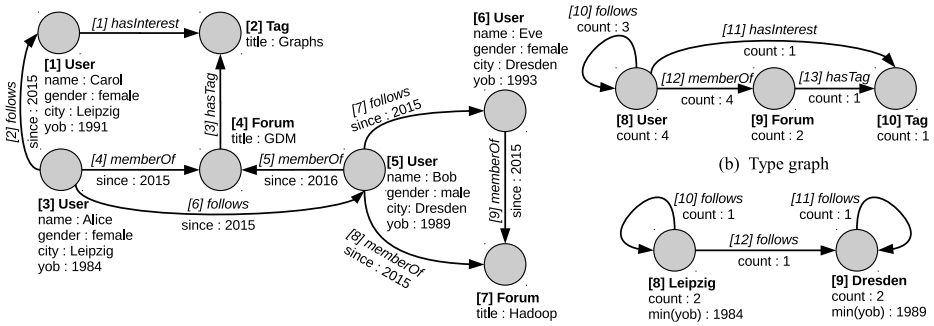


Fig. 1: (a) shows an example social network as input of the grouping operator; (b) shows the vertices and edges of (a) grouped according to their label including count aggregates stored in respective properties at the resulting vertices and edges; (c) shows the subgraph containing users and their mutual relationships grouped by users' location and edge labels including aggregate values expressing the oldest user's age per location and the number of edges among locations.

Example 1: Type Graph A simple graph analytical question is: *"Which types of entities are contained and how are they connected?"*. With regard to our social network example, the answer is shown in Figure 1(b). Here, each vertex represents a group of vertices from the original graph that share the same type. For example, vertex 8 represents the users *Alice*, *Carol*, *Bob* and *Eve*, while vertex 9 represents the forums *GDM* and *Hadoop*. Edges are grouped according to their incident vertices and their type, e.g., edge 11 represents all memberships among users and forums in the original graph. Furthermore, each vertex and edge in the summary graph stores the number of elements it represents as a new property. The resulting graph provides an overview of the underlying network and, thus, is a good starting point for more detailed analyses.

Example 2: City Graph In this example we want to group users by the city they live in, calculate the number of group members and find the smallest year of birth (yob) per group. Edges shall be grouped by their type and also being counted. To achieve this, we need to group users by the property *city* and aggregate each of these groups using the *yob* property. The resulting summary graph is shown in Figure 1(c) and reveals that our social network includes users from Leipzig and Dresden whereas the oldest person lives in Leipzig. This high level view further shows how relationships are distributed among groups.

The examples demonstrate the value of summary graphs to gain useful insights into large networks. Note that complex graph analytical questions often require the combination of multiple algorithms, e.g., Example 2 requires extracting a subgraph containing only users and their mutual relationships and replacing vertex labels by a certain property value before the summary graph can be computed.

To support analyses combining multiple techniques, we started developing Gradoop, a graph analytical system that implements the Extended Property Graph Model (EPGM) [Ju16]. The data model defines operators that can be combined to complex analytical programs on single property graphs and graph collections. Gradoop is open-source[4] and

[4] http://www.gradoop.com

built on top of Apache Flink [Al14, Ca15b], a scalable dataflow framework. Thus, the execution of the provided operators can be distributed across a cluster of machines. To our knowledge, no other distributed graph analytics framework provides a similar grouping functionality. Our main contributions can be summarized as follows:

- We formally introduce the grouping operator to flexibly compute summaries of property graphs by user-defined properties and aggregation functions.
- We discuss various analytical scenarios combining the grouping operator with other graph operators provided by the Extended Property Graph Model.
- We describe the implementation of the grouping operator in the context of Gradoop, our system for graph analytics on top of Apache Flink. We additionally introduce an optimization for unbalanced workloads.
- We present experimental results to evaluate the scalability of our implementation by applying the operator to real-world and synthetic social network data.

In Section 2, we provide the theoretical foundation of graph grouping in the context of the Extended Property Graph Model and discuss different application scenarios. Afterwards in Section 3, we describe the implementation of graph grouping utilizing operators of Apache Flink. The experimental evaluation is presented in Section 4. Finally, we discuss related work in Section 5 and conclude our work in Section 6.

2 Graph Grouping in the Extended Property Graph Model

First, we briefly introduce the used EPGM graph data model. Based thereon, we then formally define graph grouping and introduce the grouping operator in GrALa, our EPGM-specific DSL for graph analytics. We will further show how graph grouping is used to construct summary graphs either stand-alone or in combination with other graph operators.

2.1 Extended Property Graph Model

Gradoop is based on a semantically rich, schema-less graph data model called *Extended Property Graph Model* (EPGM) [Ju16]. In this model, a graph database consists of multiple possibly overlapping property graphs which are referred to as *logical graphs*. Vertices, edges and logical graphs have a type label (e.g., `User`, `memberOf` or `Community`) and may have an arbitrary set of attributes represented by key-value-pairs (e.g., `name : Alice`). Formally, an EPGM database is defined as follows:

Definition 1 *(EPGM Database). An EPGM database* $DB = \langle \mathcal{V}, \mathcal{E}, \mathcal{L}, K, T, A, \kappa \rangle$ *consists of vertex set* $\mathcal{V} = \{v_i\}$*, edge set* $\mathcal{E} = \{e_k\}$ *and a set of logical graphs* $\mathcal{L} = \{G_m\}$*. Vertices, edges and (logical) graphs are identified by the respective indices* $i, k, m \in \mathbb{N}$*. An edge* $e_k = \langle v_i, v_j \rangle$ *with* $v_i, v_j \in \mathcal{V}$ *directs from* v_i *to* v_j *and supports loops (i.e.,* $i = j$*). There can be multiple edges between two vertices differentiated by distinct identifiers. A* ***logical graph*** $G_m = \langle V_m, E_m \rangle$ *is an ordered pair of a subset of vertices* $V_m \subseteq \mathcal{V}$ *and a subset of edges* $E_m \subseteq \mathcal{E}$ *where* $\forall \langle v_i, v_j \rangle \in E_m : v_i, v_j \in V_m$*. Vertex, edge and graph properties are defined by key set K, value set A and mapping* $\kappa : (\mathcal{V} \cup \mathcal{E} \cup \mathcal{L}) \times K \rightarrow A$*. For the definition of type labels we use a label alphabet* $T \subseteq A$ *and a dedicated type property key* $\tau \in K$*.*

The EPGM defines a set of expressive operators to analyze logical graphs and collections of these. Since the in- and output of such operators are always logical graphs, the power of the

EPGM is based on the ability to combine multiple operators to graph analytical programs. For example, to achieve the summary shown in Figure 1(c), we have to extract the logical graph containing only vertices of type `User` including their mutual relationships and use the vertex property `city` as a new vertex label before applying the grouping operation. The Gradoop framework already provides operator implementations for graph pattern matching, subgraph extraction, graph transformation, set operations on multiple graphs as well as property-based aggregation and selection [Ju16].

2.2 Graph Grouping

EPGM operators are classified according to their input and output. For example, a *unary graph operator* takes a single logical graph as input and outputs either a new logical graph or a graph collection. Graph grouping belongs to the former ones as it outputs a single logical graph, which we call a *summary graph*:

Definition 2 *(Graph Grouping). For a given logical graph $G(V, E)$, a non-empty set of vertex grouping keys $K_v \subseteq K$, a set of edge grouping keys $K_\epsilon \subseteq K$ and sets of aggregate functions Λ_v and Λ_ϵ, the graph grouping operator produces a so-called summary graph $G'(V', E')$ containing super vertices and super edges. The resulting graph and its elements are added to the EPGM database, such that $\mathcal{L} \leftarrow \mathcal{L} \cup \{G'\}$, $\mathcal{V} \leftarrow \mathcal{V} \cup V'$ and $\mathcal{E} \leftarrow \mathcal{E} \cup E'$.*

Definition 3 *(Super Vertex). Let $V(G') = \{v'_1, v'_2, ..., v'_n\}$ be the vertex set of a summary graph G' and $s_v : V(G) \rightarrow V(G')$ a surjective function, then v'_i is called a super vertex and $\forall v \in V(G)$, $s_v(v)$ is the super vertex of v. Vertices in $V(G)$ are grouped based on their property values, such that for a given non-empty set of vertex grouping keys $K_v \subseteq K$, $\forall u, v \in V(G) : s_v(u) = s_v(v) \Leftrightarrow \forall k \in K_v : \kappa(u, k) = \kappa(v, k)$. A super vertex stores the properties representing the group, such that, $\forall k \in K_v, \forall u \in V(G) : \kappa(s_v(u), k) = \kappa(u, k)$.*

Definition 4 *(Super Edge). Let $E(G') = \{e'_1, e'_2, ..., e'_m\}$ be the edge set of a summary graph G' and $s_\epsilon : E(G) \rightarrow E(G')$ a surjective mapping, then e'_i is called a super edge and $\forall \langle u, v\rangle \in E(G)$, $s_\epsilon(\langle u, v\rangle)$ is the super edge of $\langle u, v\rangle$. Edge groups are determined along the super vertices and a set of edge keys $K_\epsilon \subseteq K$, such that $\forall \langle u, v\rangle, \langle s, t\rangle \in E(G) : s_\epsilon(u, v) = s_\epsilon(s, t) \Leftrightarrow s_v(u) = s_v(s) \wedge s_v(v) = s_v(t) \wedge \forall k \in K_\epsilon : \kappa(\langle u, v\rangle, k) = \kappa(\langle s, t\rangle, k)$. Analogous to super vertices, a super edge stores the properties representing the group, such that $\forall k \in K_\epsilon, \forall \langle u, v\rangle \in E(G) : \kappa(\langle s_v(u), s_v(v)\rangle, k) = \kappa(\langle u, v\rangle, k)$.*

Definition 5 *(Aggregates). Additionally, sets of associative and commutative vertex and edge aggregate functions $\Lambda_v = \{\alpha_v : \wp(V(G)) \rightarrow A\}$ and $\Lambda_\epsilon = \{\alpha_\epsilon : \wp(E(G)) \rightarrow A\}$ can be used to compute aggregated property values for super vertices and edges. The resulting value is stored at the respective super entity using a key determined by $f_\alpha : \Lambda_v \cup \Lambda_\epsilon \rightarrow K$, e.g., $\forall \alpha_v \in \Lambda_v, \forall v' \in V(G') : \kappa(v', f_\alpha(\alpha_v)) = \alpha_v(\{u \in V(G) \mid s_v(u) = v'\})$.*

In our introductory example we apply the grouping operator on a graph G representing the social network in Figure 1(a). To compute the summary graph G' of Figure 1(b), we first need to specify two sets of grouping keys. To group vertices and edges by their type label we use the type property key τ such that $K_v = K_\epsilon = \{\tau\}$. Additionally, we define two aggregate functions $\alpha_{v_{count}} : V \mapsto |V|$ and $\alpha_{\epsilon_{count}} : E \mapsto |E|$ and assign them to a property key $f_\alpha(\alpha_{v_{count}}) = f_\alpha(\alpha_{\epsilon_{count}}) =$ count.

The resulting summary graph consists of three super vertices $V(G') = \{v_8, v_9, v_{10}\}$ and four super edges $E(G') = \{v_{10}, v_{11}, v_{12}, v_{13}\}$. Considering vertex v_8 as an example one can see that

it takes the type label `User` of its underlying vertex group, i.e., $V_{v_8} = \{v_1, v_3, v_5, v_6\} \subset V(G)$, and shows an additional property (`count`) that refers to the result of the aggregate function: $\kappa(v_8, f_\alpha(\alpha_{\nu_{count}})) = \alpha_{\nu_{count}}(V_{v_8}) = 4$. Edges are grouped by the super vertices of their incident vertices as well as their label. For example, edge e_{10} represents all `follows` edges connecting vertices represented by v_8, i.e., $E_{e_{10}} = \{e_2, e_6, e_7\} \subset E(G)$. On the other hand, edge e_{12} represents all `memberOf` edges pointing from a `User` to a `Forum`. Like super vertices, super edges also store an additional property referring to the result of the aggregate function, e.g., $\kappa(e_{10}, f_\alpha(\alpha_{\epsilon_{count}})) = \alpha_{\epsilon_{count}}(E_{e_{10}}) = 3$.

2.3 Graph Grouping in GrALa

In the remainder of this paper, we will use a domain specific language called GrALa (**Gr**aph **A**nalytical **La**nguage) that has been introduced in [Ju16] to express graph analytical programs in the EPGM.[5] In GrALa, graph operators are higher-order functions that can be parameterized with user-defined functions to express custom logic, e.g., for aggregation or filtering. The operator signature for graph grouping is defined as follows:

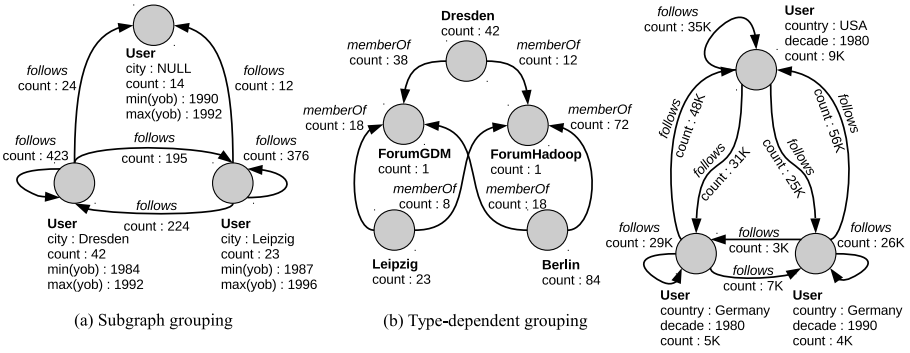
```
LogicalGraph.groupBy(
  vertexGroupingKey[], vertexAggregateFunction[],
  edgeGroupingKey[], edgeAggregateFunction[]) : LogicalGraph
```

While the first argument is a list of vertex grouping keys $K_\nu \subseteq K$, the second argument refers to a list of user-defined vertex aggregate functions Λ_ν. Analogously, the third and fourth argument are used to define edge grouping keys and edge aggregate functions. The operator returns a new logical graph that represents the resulting summary graph. The following listing demonstrates how the operator is parameterized to compute the result of our introductory example of Figure 1(b):

```
1  LogicalGraph socialNetwork = // initialize logical graph ...
2  LogicalGraph summaryGraph = socialNetwork.groupBy(
3    [:label],
4    [(superVertex, vertices -> superVertex['count'] = vertices.size())],
5    [:label],
6    [(superEdge, edges -> superEdge['count'] = edges.size())])
```

In line 1 we initialize a logical graph representing our social network either from a data source or from previously executed graph operators. Next, we declare a new variable `summaryGraph` and initialize it using the result of the grouping operator. In Figure 1(b) the social network is grouped according to type labels of vertices and edges. Therefore, we define vertex and edge grouping keys in line 3 and 5 respectively. In GrALa the symbol `:label` refers to the dedicated property key τ representing the type label. Although our example requires only a single grouping key, it is also possible to define multiple keys to further differentiate vertices and edges (e.g., `[:label, 'city']`). In lines 4 and 6 we specify anonymous aggregate functions for vertices and edges using common Lambda notation (e.g., `(parameters -> function body)`). The vertex aggregate function in line 4 counts the number of vertices represented by a super vertex. The parameters of that function are the super vertex $v' \in V(G')$ and the vertex set $\{v \in V(G) \mid s_\nu(v) = v'\}$. The size of that set is stored as a new property of the super vertex using the property key `count`.

[5] The Gradoop framework implements GrALa using the Java programming language.



(a) Subgraph grouping (b) Type-dependent grouping (c) Roll-up grouping

Fig. 2: Exemplary results for the subgraph and transformation based graph analytical programs. (a) shows a summary graph computed from a subgraph of the input graph; (b) shows a summary graph of a heterogeneous input graph; (c) shows a summary graph representing a roll-up operation

2.4 Analytical examples

A fundamental feature of the EPGM is its ability to compose graph operators to complex analytical programs. In the following, we will exemplify the abilities of the framework with a focus on graph grouping. Necessary operators will be briefly introduced, a detailed description can be found in [Ju16].

Subgraph grouping

Just like graph grouping, *subgraph* is a unary graph operator that outputs a single logical graph. The operator's arguments are user-defined predicate functions. Input vertices and edges will only be passed to the output graph if the respective predicate function evaluates to `true`. If there is only a single function defined, the operator extracts vertex-induced and edge-induced subgraphs, respectively.

The combination of subgraph and grouping operator allows the creation of partial summaries as illustrated by Figure 2(a). Script 1 lists the corresponding GrALa program. First, we extract a subgraph of a given social network that contains solely vertices of type `User` and edges of type `follows`. Therefore, we define vertex and edge predicate functions in lines 3 and 4. Both functions take a single element as input and define a condition on the corresponding type label to check if it matches the required label. During operator execution, the predicate functions are executed for each vertex and edge contained in the input graph. Since the output of the subgraph operator is a logical graph, it can be directly used as input for the grouping operator at line 5. In addition to the label, we further group vertices (users) based on the city they live in. We also provide further vertex aggregate functions to compute the minimum and maximum year of birth inside each user group. For simplicity, we use pre-defined aggregate functions provided by GrALa (e.g., `COUNT()`). In the resulting summary graph every vertex represents a group of users that share the same property value for the property key `city`. Additionally, vertices store the results of the specified aggregate functions as new properties. Since the EPGM is a schema-less data model, vertex and edge instances do not necessarily have a specified property. In Figure 2(a) this is reflected by a dedicated vertex representing all vertices without property *city*.

```
1  LogicalGraph summaryGraph = socialNetwork
2    .subgraph(
3      (vertex -> vertex[:label] == 'User'),
4      (edge -> edge[:label] == 'follows'))
5    .groupBy(
6      [:label, 'city'], [COUNT(), MIN('yob'), MAX('yob')],
7      [:label], [COUNT()])
```

Script 1: Graph grouping applied to a specific subgraph.

```
1  LogicalGraph summaryGraph = socialNetwork
2    .subgraph((edge -> edge[:label] == 'memberOf'))
3    .transform((vertex -> {
4      if (vertex[:label] == 'User') then vertex[:label] = vertex['city']
5      else vertex[:label] = vertex[:label] + vertex['title']}))
6    .groupBy([:label], [COUNT()], [:label], [COUNT()])
```

Script 2: Type-dependent grouping using vertex transformation.

With such a declarative program, an analyst is now able to explore the structure and properties of entities and their mutual relationships contained in the underlying social network. For example, one can easily see that there are more follower relations among users located in the same city than among users from different cities. The program also demonstrates the flexibility of the operator: by adding the property key `gender` to the vertex grouping keys, the super vertices in Figure 2(a) could be further divided into groups with respect to users' gender and reveal more information about the relations between gender groups from different cities.

Type-dependent grouping of heterogeneous graphs

In the previous example, we applied graph grouping to a homogeneous subgraph containing solely vertices of type `User` and edges of type `follows`. However, it might also be necessary to apply graph grouping on a heterogeneous graph, i.e., to define type-dependent vertex and edge grouping keys [YG16]. Within the EPGM, this can be achieved by combining the grouping operator and a preceding *graph transformation*. The transformation operator is a unary graph operator that allows the modification of graph, vertex and edge data whereas graph structure remains unchanged. The operator is parameterized by user-defined transformation functions that either have graph, vertex or edge data as input. Within these functions the analyst is able to modify type label and properties of the particular instance. The operator's output is a new logical graph with identical structure as the input graph but modified data of its elements. Transformation is typically helpful in data integration and ETL scenarios [Pe14a, Pe14b] or to pre-process graphs for subsequent graph operators as in our next example.

In Script 2, we create a summary graph showing different forums and their members under consideration of their cities to analyze local interests. Therefore, we first extract an edge-induced subgraph from our social network that solely contains edges of type `memberOf` since this relationship connects users with forums. In line 3, we apply a transformation function on the vertices of our previously extracted subgraph. The function modifies a vertex with respect to its current label: for users, we replace the label by the property value

```
1  LogicalGraph summaryGraph = socialNetwork
2    .subgraph((edge -> edge[:label] == 'follows'))
3    .transform((vertex -> {
4      vertex['decade'] = vertex['yob'] - (vertex['yob'] mod 10)
5      vertex['country'] = getCountry(vertex['city']) }))
6    .groupBy([:label, 'country', 'decade'], [COUNT()], [:label], [COUNT()])
```

Script 3: Graph grouping on property hierarchies using vertex transformation.

associated to their city attribute and, for forums, we concatenate original label and unique forum title to create a new label. The modified graph is then used as input for the grouping operator in line 6. Since we projected all necessary information to vertex labels, we can now group vertices and edges by label and additionally count the particular group members.

An exemplary summary graph is illustrated by Figure 2(b). The grouping operator created a vertex for each city representing users from that city. Since labels of forum vertices contain the unique forum title, the grouping operator created super vertices that represent a single vertex from the input graph. However, users in the neighborhood of these vertices and their relations to the forums have been grouped. By looking at the number of memberships stored at super edges, an analyst is now able to conclude about the interests of local user groups.

Graph grouping along property hierarchies

Another application in which transformation increases the flexibility of graph grouping is the consideration of property (or dimension) hierarchies. Here, an analyst wants to group a graph on different levels, e.g., by using a dimension hierarchy like *time* or *location*. This type of operation is also known as roll-up in data warehousing or graph OLAP scenarios [Ch08, Zh11, YWZ12]. In our social network example, users store year of birth as well as the city they live in. In Script 3, we use transformation to compute coarser levels of dimensional hierarchies by mapping years to decades (line 4) and cities to corresponding countries (line 5). The results are stored in new vertex properties and the respective property keys are used as vertex grouping keys in the subsequent grouping operator. An exemplary result of the program can be seen in Figure 2(c). In contrast to Figure 2(a), vertices now represent users that live in the same country and were born in the same decade. Using such a high-level view, the summary graph provides useful insights about a network which may contain millions or even billions of users.

3 Implementation of Graph Grouping in Gradoop

For the implementation of graph grouping and Gradoop operators in general, we have to cope with two major challenges of big data analytics: the operator's flexible integration in complex analyses as shown in our examples and its scalability for very large graphs with billions of edges. A now established approach to solve the latter problem is the massively parallel computation on shared-nothing clusters, e.g., based on the Hadoop ecosystem.

Especially promising is the utilization of distributed dataflow systems such as Apache Spark [Za12] and Apache Flink [Ca15b] that, in contrast to the older MapReduce framework [DG08], offer a wider range of operators and keep data in main memory between the execution of operators. The major challenges of implementing graph operators in these systems are identifying an appropriate graph representation, an efficient combination of

the primitive dataflow operators and the minimization of data exchange among different machines.

Our implementation of graph grouping is, like the implementation of all Gradoop operators, based on Apache Flink. A basic version of the implementation has also been contributed to Apache Flink.[6] We first give a brief introduction to Apache Flink and its programming concepts and explain the mapping of the EPGM to these concepts. We then outline the implementation of graph grouping including an optimization for unbalanced data distribution.

3.1 Apache Flink

Apache Flink [Ca15b] supports the declarative definition and execution of distributed dataflow programs. The basic abstractions of such programs are *DataSets* and *Transformations*. A DataSet is an immutable, distributed collection of arbitrary data objects, e.g., Pojos or tuple types, and transformations are higher-order functions that describe the construction of new DataSets either from existing ones or from data sources. Application logic is encapsulated in user-defined functions (UDFs), which are provided as arguments to the transformations and applied to DataSet elements.

Well-known transformations have been adopted from the MapReduce paradigm [DG08]. While the *map* transformation expects a bijective UDF that maps each element of the input DataSet to exactly one element of the output DataSet, the *reduce* transformation aggregates all input elements to a single one. Further transformations are known from relational databases, e.g., *join*, *group-by*, *project* and *distinct*. Table 1 introduces the transformations available in Apache Flink's DataSet API that are relevant for this paper. In addition, Apache Flink provides libraries for analytical tasks such as machine learning, graph processing and relational operations. To describe a dataflow, a program may include multiple chained transformations and library calls. During execution Flink handles program optimization as well as data distribution and parallel processing across a cluster of machines.

3.2 Gradoop

The Gradoop open-source library is a complete implementation of the EPGM and its operators on top of Apache Flink's DataSet API. It can be used standalone or in combination with any other library available in the Flink ecosystem. Gradoop uses three object types to represent EPGM data model elements: *graph head*, *vertex* and *edge*. A graph head represents the data associated to a single logical graph. Vertices and edges not only carry data but also store their graph membership as they may be contained in multiple logical graphs. In the following, we show a simplified definition of the respective types:

```
class GraphHead { Id; Label; Properties }
class Vertex    { Id; Label; Properties; GraphIds }
class Edge      { Id; Label; Properties; SourceId; TargetId; GraphIds }
```

Each type contains a system managed identifier (`Id`) represented by a 128-bit *universally unique identifier*[7]. Furthermore, each element has a label of type string and a set of properties. Since EPGM elements are self-descriptive, properties are represented by a key-value map whereas the property key is of type String and the property value is encoded in a byte array.

[6] https://issues.apache.org/jira/browse/FLINK-2411

[7] docs.oracle.com/javase/7/docs/api/java/util/UUID.html

Name	Description
Map	The map transformation applies a user-defined map function to each element of the input DataSet. Since the function returns exactly one element, it guarantees a one-to-one relation between the two DataSets. `DataSet<IN>.map(udf: IN -> OUT) : DataSet<OUT>`
Filter	The filter transformation evaluates a user-defined predicate function to each element of the input DataSet. If the function evaluates to `true`, the particular element will be contained in the output DataSet. `DataSet<IN>.filter(udf: IN -> Boolean) : DataSet<IN>`
Project	The projection transformation takes a DataSet containing a tuple type as input and forwards a subset of user-defined tuple fields to the output DataSet. `DataSet<TupleX>.project(fields) : DataSet<TupleY> (X,Y in [1,25])`

Join	The join transformation creates pairs of elements from two input DataSets which have equal values on defined keys (e.g., field positions in a tuple). A user-defined join function is executed for each of these pairs and produces exactly one output element.
	`DataSet<L>.join(DataSet<R>).where(leftKeys).equalTo(rightKeys)`
	`  .with(udf: (L,R) -> OUT) : DataSet<OUT>`

ReduceGroup	DataSet elements can be grouped using custom keys (similar to join keys). The ReduceGroup transformation applies a user-defined function to each group of elements and produces an arbitrary number of output elements. `DataSet<IN>.groupBy(keys).reduceGroup(udf: IN[] -> OUT[]) : DataSet<OUT>`
CombineGroup	The CombineGroup transformation is similar to a ReduceGroup transformation but is not guaranteed to process all elements of a group at once. Instead, it processes sub-groups of all elements stored in the same partition (i.e., are processed by the same worker). Thus, it can be used to decrease data volume before a ReduceGroup transformation which otherwise may require shuffling data over the network. `DataSet<IN>.groupBy(keys).combineGroup(udf: IN[] -> OUT[]) : DataSet<OUT>`

Tab. 1: Subset of Apache Flink DataSet transformations. We define `DataSet<T>` as a DataSet that contains elements of type `T` (e.g., `DataSet<String>` or `DataSet<Tuple2<Int,Int>>`).

The current implementation supports values of all primitive Java types and BigDecimal. Vertices and edges maintain their graph membership in a dedicated set of graph identifiers (`GraphIds`). Edges additionally store the identifiers of their incident vertices. To represent a graph collection, Gradoop uses a separate Flink DataSet for each element type. A logical (property) graph is a special case of a graph collection in which the graph head DataSet contains a single object:

```
class LogicalGraph {
    DataSet<GraphHead> graphHead;
    DataSet<Vertex> vertices;
    DataSet<Edge> edges;
}
```

EPGM operators are implemented using Flink transformations on a logical graph's or a graph collection's DataSets. For example, the subgraph operator is implemented using the `filter` transformation to apply user-defined predicate functions and the `join` transformation to preserve consistency in the output graph. More complex operators like graph pattern matching use a large number of different Flink transformations including iterations.

3.3 Graph Grouping

The graph grouping operator computes a summary graph by applying a series of transformations to the vertex and edge DataSets of an input graph. The algorithmic idea is to group vertices according to user-defined grouping keys and to create a super vertex for each resulting group. Then, a mapping between the original vertices and the super vertices is

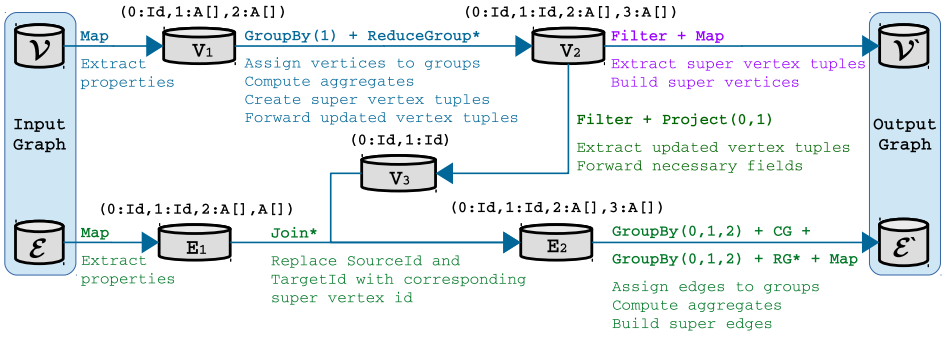


Fig. 3: Dataflow implementation of the graph grouping operator using Flink DataSets and transformations. The dataflow is subdivided into three phases: (1, blue) grouping vertices, (2, purple) building super vertices and (3, green) building super edges. Intermediate DataSets always contain tuples, whose fields can be referred to by their position, e.g., `GroupBy(1)` or `Project(0,1)`. Transformations denoted with * require inter-partition communication between workers in a cluster.

used to update source and target identifiers for each edge. Finally, edges are also grouped by their new source and target vertex and optionally by user-defined grouping keys.

Figure 3 illustrates the corresponding dataflow program from an input logical graph $G(V, E)$ to an output summary graph $G'(V', E')$. For the sake of clarity, we grouped multiple transformations (e.g., `Filter + Map`) and omitted intermediate results when possible. A comprehensive pseudocode description can be found online.[8] We use different colors to denote the three phases of the algorithm: (1, blue) grouping vertices, (2, purple) building super vertices and (3, green) building super edges. In Figure 4, we additionally illustrate a dataflow instance that computes the "Type Graph" of Figure 1(b). Using the abstraction and the concrete example, we will now discuss the three phases.

Phase 1: Grouping vertices

In a distributed dataflow framework it is important to reduce data volume whenever possible in order to avoid unnecessary network traffic. Thus, the first phase starts with mapping each vertex $v \in V(G)$ to a tuple representation containing only vertex identifier and a list of property values needed for vertex grouping and aggregation.[9] For example, in Figure 4, we map vertex v_1 to the tuple `(1,[User],[1])` as we require the vertex label for the creation of super vertices and the property value 1 to compute the `count` aggregate. Note that grouping and aggregate values need to be ordered. Vertices (and edges) without a required grouping value show the `Null`-value instead. Missing aggregate values are replaced by a default value determined by the particular aggregate function.[10] In Figures 3 and 4, the intermediate result containing vertex tuples is represented by DataSet V_1.

In the second step of phase 1, vertex tuples are grouped by the previously extracted grouping values (position 1 inside the tuple). Each group is then processed by a ReduceGroup function

[8] http://dbs.uni-leipzig.de/file/grouping_pseudocode.pdf

[9] In Figure 3, lists of property values are denoted by the type `A[]`.

[10] The `count` aggregate function is implemented as a special case of the `sum` aggregate function. In that case, the property value "1" is added automatically.

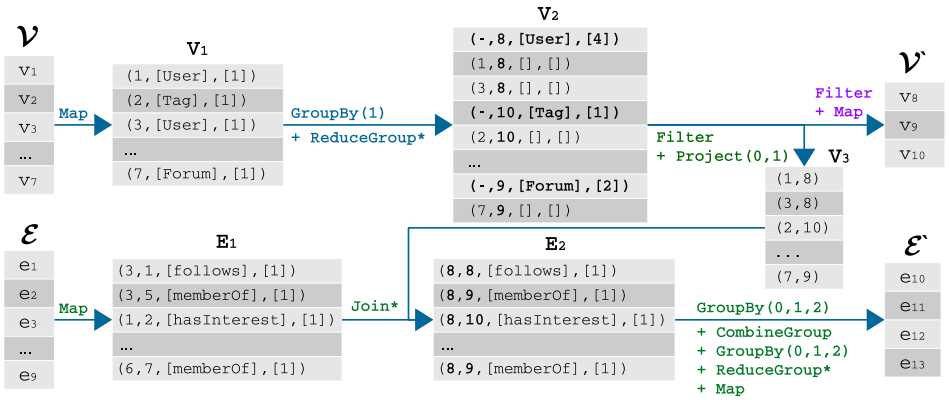


Fig. 4: Dataflow using the graphs from Figure 1(a) as input and Figure 1(b) as output.

which has two main tasks: (1) creating a *super vertex tuple* for each group and (2) creating a map between original vertex id and corresponding super vertex id. A super vertex tuple consists of a new vertex id, the property values representing the group and the results of the provided aggregate functions. In Figure 4, the intermediate DataSet V_2 contains three highlighted super vertex tuples, one for each vertex label in the input graph. For example, the tuple `(-,8,[User],[4])` represents all vertex tuples with a grouping value `User` including the result of the `count` aggregate function (`4`). Additionally, the ReduceGroup function outputs a super vertex-mapping for every member, e.g., `(1,8,[],[])` is derived from tuple `(1,[User],[1])` as vertex `1` belongs to super vertex `8`.

Phase 2: Building super vertices

In this phase, we construct the final super vertices V' from the previously created super vertex tuples. Therefore, we need to filter the tuples from the intermediate DataSet V_2 and use a map transformation to construct a new `Vertex` instance for each tuple. In Figure 4, the super vertex tuple `(-,8,[User],[4])` is mapped to super vertex v_8 of Figure 1(b), which stores the grouping and aggregate values as new properties.

Phase 3: Building super edges

In the last phase, we update the edges of input DataSet E according to their super vertices and group them to super edges E'. Similar to phase 1, we first reduce data volume by mapping all edges $e \in E$ to necessary information. In DataSet E_1, each edge is represented by a tuple containing its source and target id as well as required grouping and aggregate values. For the next step, we also require the super vertex-mappings from the intermediate DataSet V_2. The mappings are extracted using a filter transformation and we further reduce their memory footprint by projecting only necessary fields. DataSet V_3 now represents a complete mapping between original vertex id and super vertex id. In the example of Figure 4, vertex ids 1, 3, 5 and 6 are associated with vertex id 8, while 2 is represented by vertex id 10. We now join DataSets E_1 and V_3 on the original vertex ids to update the source and target vertex id for each edge tuple. Technically, we require separate join operations for both identifiers (see pseudocode for details). DataSet E_2 represents the updated edge tuples, e.g., tuple `(3,1,[follows],[1])` is updated to `(8,8,[follows],[1])`, since source id (3) and target

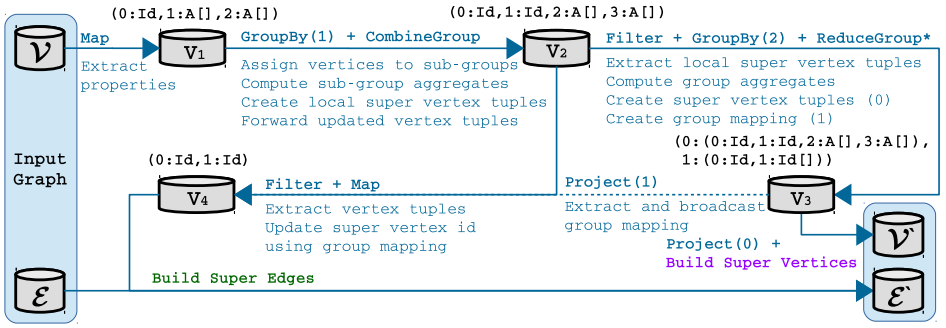


Fig. 5: Optimized dataflow for Phase 1.

id (1) are both represented by super vertex id 8. In contrast, tuple `(3,5,[memberOf],[1])` is mapped to `(8,9,[memberOf],[1])` as source and target vertex belong to different super vertices (i.e., users and forums).

Since the updated tuples in E_2 are logically connecting super vertices, we can group them by source and target id as well as their grouping property values. The creation of *super edge tuples* is done in two consecutive steps. First, we use a CombineGroup transformation to group edge tuples per data partition. Depending on the data distribution, each partition may contain multiple sub-groups. For each of these sub-groups, the CombineGroup function produces exactly one *local super edge tuple* that stores the grouping values and the results of the aggregate functions for that sub-group. After the CombineGroup step, there might be two tuples representing the same edge group, e.g., `(8,8,[follows],[2])` and `(8,8,[follows],[1])`. To create exactly one tuple for each edge group, we again need to group the local super edge tuples by the same fields, but this time followed by a ReduceGroup function to guarantee that all tuples representing the same edge group are processed together. Since the computation logic of both functions is identical, we can use the same UDF in the combine and reduce step. The output is one super edge tuple, e.g., `(8,8,[follows],[3])`, that represents the whole group. Finally, each super edge tuple is mapped to a new `Edge` that stores grouping and aggregate values as new properties.

After phase 3, the computed super vertices V' and edges E' are used as parameters to instantiate a new logical graph G'. During instantiation, a new `GraphHead` is created and graph memberships of super vertices and edges are updated. The logical graph is then returned to the program and can be sent to either a data sink or subsequent operations.

3.4 Handling unbalanced workloads

In distributed computing, the application of a local combination step is generally useful as it potentially reduces network traffic [DG08, Ma10]. However, the efficiency of the combine step depends on the physical data distribution: if each worker has one element of each group, the CombineGroup function has no effect. Nevertheless, it is advisable to add a combine step if the computation logic permits it. In phase 3, we use a CombineGroup transformation to map edge sub-groups to local super edge tuples which are then shuffled across the network in the subsequent ReduceGroup transformation. For edges, this step is straightforward as each sub-group can be processed independently. However, for vertices, the algorithm needs

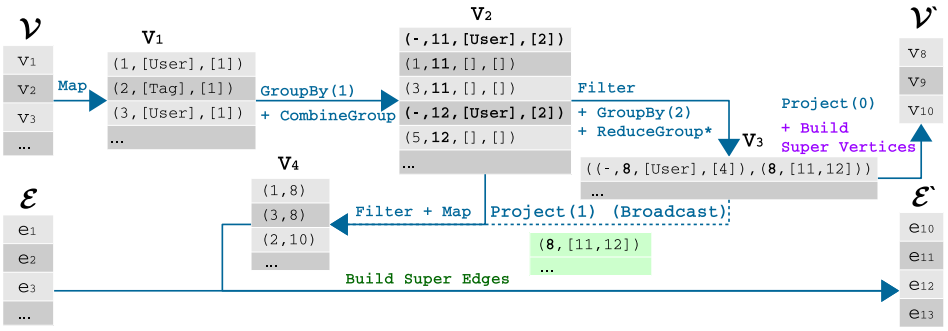


Fig. 6: Optimized dataflow using the graphs from Figure 1(a) as input and Figure 1(b) as output.

to guarantee that all vertices of a group are assigned to the same super vertex. In Figure 4, this constraint is met by applying a ReduceGroup function directly on the grouped vertex tuples. Yet, this leads to an unbalanced workload if the group size distribution is skewed since all group members need to be transferred to the same worker.

Figure 5 illustrates an alternative approach for phase 1. We again start with building vertex tuples using a map transformation. We then use a CombineGroup transformation on the grouped vertex tuples which creates DataSet V_2. The UDF logic is identical to the original phase, however, we now create *local* super vertex tuples potentially representing the same vertex group. Figure 6 shows an example: V_2 contains two local super vertex tuples representing vertices of type `User` and mapping tuples contain local super vertex ids. We now filter local super vertex tuples from V_2. Up to this point, the dataflow does not require any data shuffling over the network. The reduced DataSet is then grouped again by the grouping values and processed in a ReduceGroup transformation. Here, we apply a different logic: in addition to the final super vertex tuple, we also create a mapping from the final super vertex id to all local super vertex ids representing the same group. In Figure 6, DataSet V_3 contains tuples composed of two tuples: the super vertex tuple (e.g., `(-,8,[User],[4])`) and the respective group mapping (e.g., `(8,[11,12])`).

We extract the super vertex tuples from the composed tuples of V_3 using projection and create final super vertices in phase 2. Prior to phase 3, we need to update the mappings between vertices and local super vertices in V_2. After filtering these tuples from V_2, we replace the local super vertex id by the global one in a map transformation. To achieve this, we use a Flink feature called *broadcasting*, which allows distributing an entire DataSet to all workers in a cluster and reading it in a UDF context. In Figure 6, we highlighted the DataSet that is being broadcasted to the map function. In this function, we just need to determine the super vertex id which maps to the current local super vertex id of the tuple. In the example of Figure 6, the vertex tuple `(1,11,[],[])` is updated to `(1,8)` since the local super vertex id 11 is represented by super vertex id 8. The resulting DataSet V_4 is identical to DataSet V_3 in Figure 3 and is used to create super edges in the final phase.

In contrast to regular ones, broadcast DataSets are kept in-memory on each worker. Since the size of group mappings depends on the number of vertex groups and data distribution, memory consumption of the broadcast DataSet may vary heavily. Before applying the broadcasting approach, one needs to consider that computation will stop if a broadcast DataSet exceeds the available main memory of a worker.

Name	$\|V\|$	$\|E\|$	Disk size	$\|T_V\|$	$\|T_E\|$	$\|V_{\tau=Person}\|$	$\|E_{\tau=knows}\|$
GA.10	260 K	16.6 M	4.5 GB	8	8	235 K (90.2%)	10.2 M (61.2%)
GA.100	1.7 M	147 M	40.2 GB	8	8	1.67 M (98.4%)	101 M (68.9%)
GA.1000	12.7 M	1.36 B	372 GB	8	8	12.67 M (99.8%)	1.01 B (74.4%)
Pokec	1.6 M	30.6 M	5.6 GB	1	1	1.6 M (100%)	30.6 M (100%)

Tab. 2: Statistics of the social network datasets used in the benchmarks.

4 Evaluation

In the experiments we evaluate the scalability of our implementation with respect to increasing computing resources and data volume. We further analyze the operators' runtime according to the number of grouping keys and aggregate functions. Finally, we study the effect of a combiner in the vertex grouping phase as described in Section 3.4.

4.1 Experimental setup

We perform our experiments using datasets generated by the *Graphalytics* (GA) benchmark for graph processing platforms [Er15, Ca15a]. The generator creates heterogeneous social networks with a schema similar to our examples and structural characteristics like those of real-world networks: node degree distribution based on power-laws and skewed property value distributions [Er15]. We additionally use the Pokec social network[11] containing users including their properties and mutual friendship relations. Table 2 provides an overview about the used datasets. Since several experiments use GA subgraphs solely containing vertices of type *User* and edges of type *knows*, we include their respective share in the table.

Table 3 shows the different configurations of the grouping operator used in the experiments. Configurations 1 to 4 compute type graphs and are mainly used in the scalability experiments. In configurations 5 to 13, we use a varying numbers of vertex and edge grouping keys as well as none, one or multiple aggregate functions. The used datasets either explicitly or implicitly fulfill the specified properties. For example, in Graphalytics, a users' location is encoded by an edge to a vertex of type `City`. We thus align all datasets to a common schema in a preceding ETL step.

The benchmarks are performed on a cluster with 16 worker nodes. Each worker consists of an E5-2430 6(12) 2.5 Ghz CPU, 48 GB RAM, two 4 TB SATA disks and runs openSUSE 13.2. The nodes are connected via 1 Gigabit Ethernet. Our evaluation is based on Hadoop 2.6.0 and Flink 1.0.3. We run Apache Flink standalone with 6 threads and 40 GB memory per worker. In our experiments, we vary the number of workers by setting the parallelism parameter to the respective number of threads (e.g., 2 workers correspond to 12 threads). The datasets are stored in HDFS (default settings) using a Gradoop specific JSON format and distributed using hash-based partitioning. We tested both operator implementations described in the preceding section either using a ReduceGroup function (RG) or an additional CombineGroup function (CG). Runtimes are reported by Flink's execution environment and include reading the input graph from HDFS and writing the summary graph to HDFS. In the subsequent results, each datum represents the average runtime of five executions.

[11] https://snap.stanford.edu/data/soc-pokec.html

Config.	Vertex keys	Vertex aggregate functions	Edge keys	Edge aggregate functions
1	:label	-	-	-
2	:label	COUNT()	-	-
3	:label	COUNT()	:label	-
4	:label	COUNT()	:label	COUNT()
5	city	-	-	-
6	city	COUNT()	-	-
7	city, gender	-	-	-
8	city, gender	COUNT()	-	-
9	city, gender	COUNT(), MIN(yob)	-	-
10	city, gender	COUNT(), MIN(yob), MAX(yob)	-	-
11	city	COUNT()	-	COUNT()
12	city	COUNT()	-	COUNT(), MIN(since)
13	city	COUNT()	-	COUNT(), MIN(since), MAX(since)

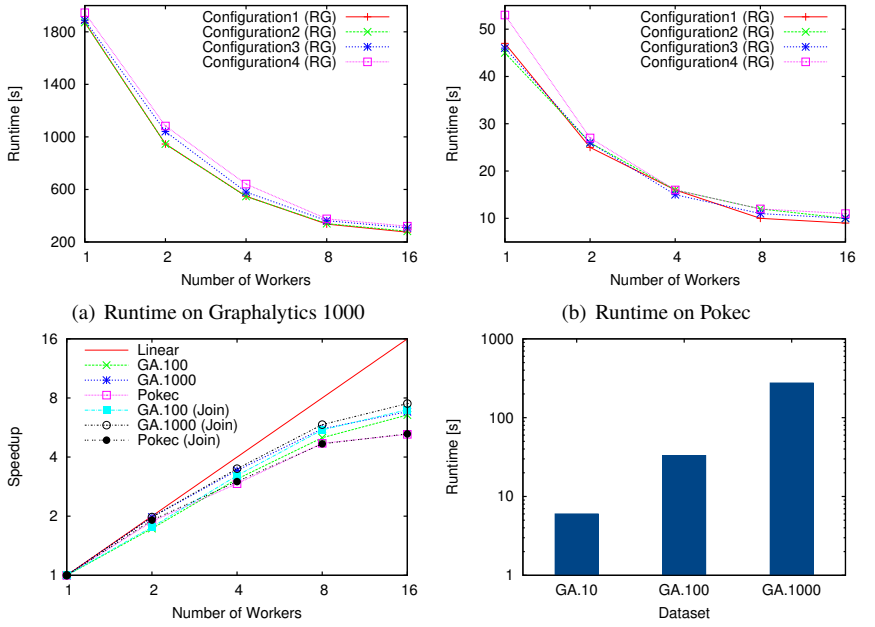
Tab. 3: Different configurations for the grouping operator.

4.2 Experimental results

Scalability We first evaluate absolute runtime and relative speedup of our implementation using configurations 1 to 4 on Graphalytics 100, 1000 and Pokec. We execute the operator on each dataset using an increasing number of workers for each run. The runtime results for Graphalytics 1000 and Pokec are shown in Figure 7(a) and 7(b), respectively. For the largest graph with 1.3 B edges, we could decrease the runtime from about 30 minutes on a single machine to 4.5 minutes on 16 workers. For the real-world network with 30 M edges, we could reduce runtimes to only 10 seconds. Using configuration 4, the execution requires the most time due to the highest data volume and computational cost caused by type labels and aggregates. However, on both datasets, the runtime is close to configurations 1 to 3. Figure 7(c) illustrates the relative speedup for configuration 1 including Graphalytics 100. One can see that up to four workers, the speedup is nearly linear and degrades after this. We assume that this is due to the two data intensive join transformations in phase 3. Here, the edge tuples need to be shuffled across the cluster which makes the network the limiting resource (also because of its limited bandwidth of only 1 Gbps). In Figure 7(c), we added the speedup solely for the join operation which aligns with the speedup of the complete runtime and thus verifies our assumption.

We also evaluated scalability with increasing data volume and a fixed number of workers. The results in Figure 7(d) show that the runtime increases almost linearly with the data volume. Using configuration 1, the operator execution required about 30 seconds on GA.100 (40 GB) and 275 seconds on GA.1000 (372 GB).

Operator parametrization In configurations 5 to 13, we parameterized the operator with a varying number of grouping keys and aggregate functions. We execute the operator on the `User`-subgraphs of GA.100 and GA.1000 as well as on the Pokec dataset using 16 workers. The results are shown in Figure 8. The first observations are that the execution times for all configurations vary only slightly and scale almost linearly from GA.100 to GA.1000. By looking at config. 5 and 7 (or 6 and 8), we can also observe that a higher number of vertex grouping keys only leads to a small increase in runtime (e.g., 319 seconds for config. 5 and 350 seconds for config. 7 on GA.1000.SG). The increase is caused by a higher number of



(a) Runtime on Graphalytics 1000

(b) Runtime on Pokec

(c) Speedup for configuration 1 (RG)

(d) Runtime for configuration 1 (RG) on 16 workers

Fig. 7: Evaluation results for the scalability benchmarks.

resulting super vertices and edges, for example, config 5 on GA.1000.SG leads to 1149, while config. 7 leads to 2298 super vertices.

In configurations 7 to 10 we increased the number of vertex aggregate functions and in configurations 11 to 13 the number of edge aggregate functions. The latter could only be executed on Graphalytics since Pokec has no edge properties. The results in Figure 8 show that a higher number of aggregate functions does not lead to a higher runtime. This is due to the fact, that in our Reduce- and CombineGroup functions, all aggregates are computed in a single iteration over the vertex and edge tuples, respectively.

Optimization for data skew We finally study the effect of an additional combination step in the vertex grouping phase. Since the type label distributions of our datasets are skewed [Er15], we use configurations 1 to 4 to compute type graphs on 16 workers. The results are shown in Figure 9. We executed the operator with each configuration using a ReduceGroup function (RG) and an additional CombineGroup function (CG). For the synthetic datasets, the benefit is generally small but more distinctive for the larger dataset. However, for the real-world dataset, we could achieve an average runtime reduction of about 15%. We believe that this is caused by the fact, that Pokec contains only one vertex type. Thus, in the ReduceGroup implementation, one worker needs to process all vertices, while in the CombineGroup implementation, load distribution and network traffic are improved due to a local preprocessing of vertices. For the synthetic datasets this effect is not that strong since the load distribution is generally better due to the higher number of type labels, i.e., vertex groups are processed by multiple workers.

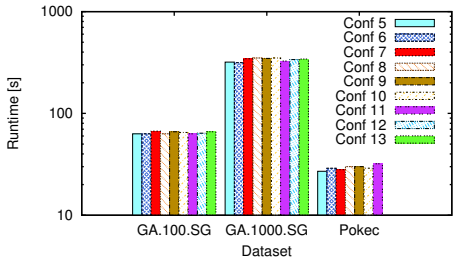


Fig. 8: Runtime for different configurations.

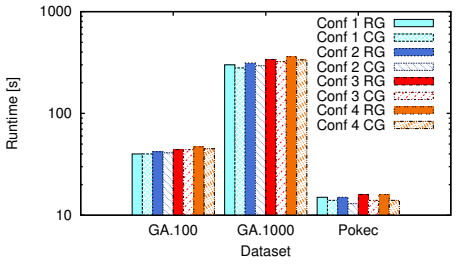


Fig. 9: Comparison of vertex grouping phases.

5 Related Work

Gradoop in general is related to graph processing frameworks on top of distributed dataflow systems, e.g., GraphX on Apache Spark [Xi13] and Gelly on Apache Flink [Ca15b]. These libraries focus on the efficient execution of iterative graph algorithms. However, in contrast to the EPGM, the implemented graph data models are generic, which means arbitrary user-defined data can be attached to vertices and edges. In consequence, model-specific operators, e.g., graph grouping, need to be user-defined, too. Hence, using those libraries to solve complex analytical problems becomes a laborious programming task. In contrast, Gradoop targets the data scientist by providing an abstraction from the underlying framework, in particular an expressive data model and declarative operators.

Graph grouping is related to the area of online analytical processing (OLAP) on graphs. Here, attributes of the graph elements are considered as dimensions and a summary graph (also denoted by aggregate graph) is one of many cuboids in a so-called graph cube. Most of the publications focus on constructing and querying graph cubes from homogeneous or heterogeneous input graphs. GraphOLAP [Ch08] first discusses grouping of single graphs and roll-up/drill-down and slice/dice operations by overlaying and filtering graphs, respectively. In [THP08], the authors introduce SNAP, an algorithm to construct summary graphs based on user-defined vertex keys and relationship types, and k-SNAP, which produces k super vertices by separating vertices depending on structural similarity. GraphCube [Zh11] extends the concepts of [Ch08] by the definition of crossboid queries enabling analysis through different levels of graph aggregations. While the previous approaches focus on homogeneous [Ch08, THP08, Zh11], vertex-attributed [THP08, Zh11] input graphs, HMGraph [YWZ12] and GRAD [Gh15] enable analyzing heterogeneous, vertex-attributed [YWZ12] graphs. HMGraph introduces additional operators on vertex-dimensions while GRAD proposes an extension to the property graph model that simplifies the definition of hierarchies in a graph cube. In [YG16], the authors introduce type-dependent grouping of heterogeneous information networks and, like [THP08], also group vertices based on a similarity function using graph entropy.

There are two previous approaches to compute graph cubes in a distributed fashion: Distributed GraphCube [DGS13] on Apache Spark and Pagrol [Wa14] on Hadoop MapReduce. While both are distributed implementations of GraphCube and thus work only on homogeneous graphs, Pagrol additionally considers edge attributes as dimensions. The authors show

that their implementations scale for the computation of the complete graph cube [Wa14] as well as single cuboids [DGS13].

In contrast to the existing Graph OLAP approaches, the graph grouping operator of Gradoop is not focusing on the creation of a graph cube containing all possible summary graphs/cuboids of a heterogeneous network. Instead, we built a framework that not only focuses on graph grouping, but also allows the flexible integration of summary graphs in combination with other complex graph operations. With regard to real-world data, which is typically semi-structured and highly dynamic, we consider our approach to be advantageous in comparison to pre-computing a complete graph cube. Furthermore, our operator provides user-defined aggregation functions, is able to handle semi-structured, heterogeneous data and allows for the interactive computation and exploration of summary graphs.

6 Conclusion and Future Work

We introduced a graph operator for the efficient, distributed grouping of large-scale, semi-structured property graphs. As the operator is part of the Extended Property Graph Model, it can be flexibly supplemented with other graph operators to express various analytical problems. Operator implementations have been contributed to the Gradoop graph analytics framework as well as to Apache Flink. One major challenge was the efficient mapping of graph grouping semantics to the abstractions provided by Flink's batch API and, at the same time, considering the absence of shared memory and the reduction of network traffic. In our experimental evaluation, we could demonstrate that our implementation scales well with graph size and can achieve very low response times on real-world networks which is a first step towards interactive exploration of large, distributed graphs. We could also show that the implementation handles skewed data distributions by leveraging Flinks combiner and broadcasting capabilities. However, our experiments revealed that a major limitation for scalability is data shuffling across the cluster. To further improve scalability of graph grouping and Gradoop operators in general, we are looking into different graph representations and graph partitioning strategies.

Besides runtime optimization, we see multiple directions for future work. First, as targeted users of Gradoop are data scientists, one goal is to improve our DSL's declarativity, e.g., to explicitly support type-dependent grouping or OLAP operations. Furthermore, we aim to add new features required by graph OLAP scenarios, for example, efficient roll-up/drill down operations and a distributed caching mechanism for summary graphs. Finally, since many real-world graphs are highly dynamic, i.e., their structure changes over time, we will investigate in implementing graph operators on dynamic graphs or rather graph streams. Here, the major challenge is the definition of a graph stream model and operator semantics.

7 Acknowledgments

This work is partially funded by the German Federal Ministry of Education and Research under project ScaDS Dresden/Leipzig (BMBF 01IS14014B).

References

[Al14] Alexandrov, A. et al.: The Stratosphere Platform for Big Data Analytics. The VLDB Journal, 23(6), December 2014.

[An12] Angles, R.: A Comparison of Current Graph Database Models. In: Proc. ICDEW. 2012.

[Ca15a] Capotă, M. et al.: Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In: Proc. GRADES. 2015.

[Ca15b] Carbone, P. et al.: Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull., 38(4), 2015.

[Ch08] Chen, C.; Yan, X.; Zhu, F.; Han, J.; Yu, P. S.: Graph OLAP: Towards online analytical processing on graphs. In: Proc. ICDM. 2008.

[DG08] Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM, 51(1), January 2008.

[DGS13] Denis, B.; Ghrab, A.; Skhiri, S.: A distributed approach for graph-oriented multidimensional analysis. In: Proc. Big Data. Oct 2013.

[Er15] Erling, O. et al.: The LDBC Social Network Benchmark: Interactive Workload. In: Proc. SIGMOD. 2015.

[Gh15] Ghrab, A.; Romero, O.; Skhiri, S.; Vaisman, A.; Zimányi, E.: A Framework for Building OLAP Cubes on Graphs. In: Proc. ADBIS. 2015.

[Ju16] Junghanns, M.; Petermann, A.; Teichmann N.; Gómez K.; Rahm E.: Analyzing Extended Property Graphs with Apache Flink. In: Proc. SIGMOD NDA Workshop. 2016.

[Ma10] Malewicz, G. et al.: Pregel: A System for Large-scale Graph Processing. In: Proc. SIGMOD. 2010.

[Ne10] Newman, M.: Networks: An Introduction. 2010.

[Pa11] Pavlopoulos, G. A. et al.: Using graph theory to analyze biological networks. BioData Mining, 4(1), 2011.

[Pe14a] Petermann, A.; Junghanns, M.; Müller, R.; Rahm, E.: BIIIG: Enabling business intelligence with integrated instance graphs. In: Proc. ICDE Workshops. 2014.

[Pe14b] Petermann, A.; Junghanns, M.; Müller, R.; Rahm, E.: Graph-based Data Integration and Business Intelligence with BIIIG. PVLDB, 7(13), 2014.

[RN10] Rodriguez, M. A.; Neubauer, P.: Constructions from Dots and Lines. arXiv:1006.2361v1, 2010.

[THP08] Tian, Y.; Hankins, R. A.; Patel, J. M.: Efficient Aggregation for Graph Summarization. In: Proc. SIGMOD. 2008.

[Wa14] Wang, Z. et al.: Pagrol: Parallel graph olap over large-scale attributed graphs. In: Proc. ICDE. 2014.

[Xi13] Xin, R. S.; Gonzales, J. E.; Franklin, M. J.; Stoica, I.: GraphX: A Resilient Distributed Graph System on Spark. In: Proc. GRADES. 2013.

[YG16] Yin, D.; Gao, H.: A flexible aggregation framework on large-scale heterogeneous information networks. Journal of Information Science, 2016.

[YWZ12] Yin, M.; Wu, B.; Zeng, Z.: HMGraph OLAP: A Novel Framework for Multi-dimensional Heterogeneous Network Analysis. In: Proc. DOLAP. 2012.

[Za12] Zaharia, M. et al.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. NSDI. 2012.

[Zh11] Zhao, P.; Li, X.; Xin, D.; Han, J.: Graph Cube: On Warehousing and OLAP Multidimensional Networks. In: Proc. SIGMOD. 2011.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 123

The STARK Framework for Spatio-Temporal Data Analytics on Spark

Stefan Hagedorn[1] Philipp Götze[1] Kai-Uwe Sattler[1]

Abstract: Big Data sets can contain all types of information: from server log files to tracking information of mobile users with their location at a point in time. Apache Spark has been widely accepted for Big Data analytics because of its very fast processing model. However, Spark has no native support for spatial or spatio-temporal data. Spatial filters or joins using, e.g., a *contains* predicate are not supported and would have to be implemented inefficiently by the users. Also, Spark cannot make use of, e.g., spatial distribution for optimal partitioning. Here we present our STARK framework that adds spatio-temporal support to Spark. It includes spatial partitioners, different modes for indexing, as well as filter, join, and clustering operators. In contrast to existing solutions, STARK integrates seamlessly into any (Scala) Spark program and provides more flexible and comprehensive operators. Furthermore, our experimental evaluation shows that our implementation outperforms existing solutions.

1 Introduction

Spark has been widely accepted as the data processing platform for big data sets providing better performance than classic Hadoop MapReduce as well as a more powerful expressiveness due to its large set of operators, support for cyclic data flow, and language-integrated queries. Spark supports a rather general data model, which allows to easily create programs for processing any type of data. Typically, data is loaded as text files and converted into their respective type of the chosen programming language or custom classes to represent complex data types.

One important class of complex data types are spatio-temporal data. For example, such data is created by sensors that record the location or movement of users or objects that periodically announce their current position. Another example for spatio-temporal data are *event data*, describing "something that happens at some place at some time". Event data is not generated by sensors only, but can also be found in many textual sources like news articles, blogs, tweets, and social media. Here, so-called taggers detect, extract, and normalize spatial and temporal expressions from the sources and prepare the event data for further analysis steps.

A typical use case for analyzing spatio-temporal event data is to find correlated events in terms of their spatial and/or temporal components, i.e. if they are close to each other – or to a reference object – in time and space. Example applications are among others recommenders for events (e.g., in the cultural domain or for location-based services), mining crime data

[1] TU Ilmenau, Databases & information systems Group, Ilmenau, Germany, *first.last*@tu-ilmenau.de

(e.g., for predictive policing or homeland security) or political news (e.g., from the GDELT data set).

When event data is extracted in an automated process from large document repositories or the Web, event analysis is an interactive and exploratory process dealing with huge and often unknown data sets. Loading, indexing, and processing this data in a spatial or relational DBMS is often too time-consuming and requires a lot of preprocessing. Instead, tools like Apache Spark can directly process this data and can be used to implement complex analytical pipelines step-by-step. However, because Spark does not provide native support for spatial or temporal data, users have to define custom classes to represent spatial and/or temporal features and additionally implement special operators for spatio-temporal processing. Furthermore, efficient processing of large data sets requires to exploit data parallelism by partitioning the data accordingly, e.g., on its proximity, which is also not supported directly in Spark.

In order to address these challenges, a few extensions to Spark have been proposed recently. However, these solutions lack in an intuitive and integrated DSL, in support for spatio-temporal instead of only spatial data, as well as in a comprehensive and complete set of operators.

Our contribution is the following:

1. We present the STARK[2] framework for spatio-temporal data processing,
2. that provides an intuitive DSL and seamless integration with Apache Spark.
3. STARK includes an expressive set of operators, including filters, joins, and clustering,
4. and it supports spatial partitioning and indexing for efficient computations.

The remainder of the paper is organized as follows. After a discussion of related work in Sect. 2, we identify important requirements of spatio-temporal processing in Spark in Sect. 3. We then describe the architecture of the STARK framework in Sect. 4, followed by details for partitioning and indexing in Sect. 5. In Sect. 6 we show the internals of the spatial operators. Results of our experimental evaluation and comparison with GeoSpark and SpatialSpark are presented in Sect. 7. Finally, we conclude the main results of our work in Sect. 8 and point out to future work.

2 Related Work

Spatial data support has been implemented in presumably every type of data storage and processing system. Traditional relational database systems have built-in geometry data types and operations: Oracle's DBMS has data types for, among others, points, lines, and polygons, which can be defined using the `SDO_GEOMETRY` type. The system uses R-trees for indexing and supports *within distance*, *nearest neighbors* and other types of queries [Or14]. IBM DB2 contains a spatial extender which provides various spatial data types that all share a parent type called `ST_GEOMETRY`. It supports a Grid Index where the grid cells are indexed

[2] https://github.com/dbis-ilm/stark

using a B-tree [IBM13]. Microsoft SQL Server provides similar functionality and also uses a hierarchical grid index. The open source systems MySQL and PostgreSQL also support spatial data, where PostgreSQL uses the PostGIS extension, and support indexing using R-trees.

With the advance of Big Data, the MapReduce paradigm and its open source implementation in Hadoop became very popular and support for geospatial operators and indexing was needed on this platform as well. In [WP+14] Whitmann et al. present a framework to index spatial data for the Hadoop platform that uses quadtrees to support spatial queries. On each node, a partial tree is created which is then shuffled to other nodes and combined to a subtree of a complete index.

The first approach to implement spatial operations as an extension for Hadoop MapReduce is SpatialHadoop [EM13; EM15]. The framework provides spatial operators for range queries, k nearest neighbors, and joins. SpatialHadoop employs two index levels: on a global level an index partitions data across all nodes while a second index organizes data inside each partition. These indexes are used on read to eliminate records that do not contribute to the final result. As index structures, SpatialHadoop supports grid files, R-tree, and R+-trees.

Another approach that extends the plain Hadoop MapReduce framework with spatial operators is HadoopGIS [AW+13]. Similarly to SpatialHadoop, HadoopGIS utilizes a two level indexing: a global partition indexing and an optional local spatial indexing. The query processing engine, called RESQUE, uses these indexes to identify partitions to load and to speed up processing the required partitions. The RESQUE engine provides spatial operators like *intersects*, *contains*, *distance*, etc. The HadoopGIS system is integrated into Hive to provide a declarative SQL-like query language as user interface.

Accumulo[3] is a key-value store on top of Hadoop and is based on the design on Google's BigTable. In GeoMesa [F+13] the keys are created as a combination of the temporal value and the Geohash[4] representation of the spatial component. It is primarily designed for point data and non-point data has to be decomposed into multiple disjoint geohashes, resulting in duplicated entries in the index. It seems that data always has to have a spatial and a temporal component. When querying data, only those data items are considered, that intersect with the query region, based on the computed geohashes. GeoWave [Na] is a geospatial index that is also based on Accumulo or HBase. Like GeoMesa, it uses Space Filling Curves to represent multidimensional objects as 1-dimensional keys.

The in-memory execution model of Spark became very popular as it reduces the execution time dramatically, compared to MapReduce jobs. Currently, there are two systems that implement spatial operators for Spark: GeoSpark and SpatialSpark.

GeoSpark [YWS15; YWS16] is a Java implementation that comes with four different RDD types: `PointRDD`, `RectangleRDD`, `PolygonRDD`, and `CircleRDD`. These special RDDs internally maintain a plain Spark RDD that contains elements of the respective type, i.e. points, rectangles, polygons, and circles. GeoSpark supports k nearest neighbor queries,

[3] https://accumulo.apache.org
[4] https://en.wikipedia.org/wiki/Geohash

range queries, and join queries with *contains* or *intersects* predicates and each of these queries can be executed with or without using an index. The predicate *withinDistance* is only supported for joins. As described in [YWS15], GeoSpark supports R-trees and quadtrees to create an ad hoc index the RDDs. However, during the evaluation it showed that choosing quadtrees is not implemented. A persistent index does not seem to be possible since there is no index load functionality. Internally, GeoSpark uses the JTS library[5] which also provides the index structures. JTS does not support nearest neighbor queries on the indexes, though, and thus, GeoSpark uses their own extension to JTS, called JTSplus[6], to support these type of queries. GeoSpark comes with several partitioning techniques: R-Tree partitioning as well as Voronoi, Hilbert, and fixed grid partitioning. The main drawbacks of GeoSpark are that their spatial RDDs can only hold geometries of one certain type. On the one hand, this makes it impossible to load a data set that contains different geometry types in one column and on the other hand all other columns are removed when putting the data into these spatial RDDs. This also means that it is not possible to process the data in subsequent steps since related columns such as an ID are not available anymore. Furthermore, GeoSpark only has support for spatial data and temporal aspects cannot be modeled or processed. From a user perspective, GeoSpark provides an API which does not integrate tightly into Spark. RDDs are not created by transformations or actions, but by creating new objects and passing in values. Also operations like joins are not implemented as functions on these RDDs, but as extra classes. This way, the user has to adopt yet another API.

The goal of the SpatialSpark approach described in [YZG15] is to provide a parallel join technique for large spatial data sets with main focus on parallel hardware like multi-core CPUs and GPUs. To compute a join, the complete right relation is indexed using an R-tree which is then made available to all worker nodes using Spark's broadcast variables. After that, all items of the left relation are probed against that R-tree to find join partners. If the right relation does not fit into memory, SpatialSpark provides Fixed Grid Partitioning, Binary Space Partitioning, and Sort Tile Partitioning with and without using an R-tree as index [YZG]. As filter operation, SpatialSpark supports range queries with the predicates *contains*, *whithin* (*containedBy*), and *withinDistance*. The partitioning techniques from above, however, can not be used for filter operations. When querying a persistent index for these range queries the *intersects* predicate is compulsorily used. On top of that, k nearest neighbor queries are not possible. While SpatialSpark provides spatial operations for Spark, the core work of the authors is to integrate spatial operations into Impala. Thus, there is no real API and many things have to be done still by hand. Internally, they expect RDDs with an ID and a geometry object, which are processed when calling the specific query object (like `RangeQuery` or `BroadcastSpatialJoin`). Similar to GeoSpark, no other payload but the ID is allowed and, furthermore, the result of a join returns only the matched pairs IDs, which requires additional joins afterwards to retrieve the complete tuple in the application.

In Tab. 1 we outline the key differences from our STARK approach to the two frameworks for Spark GeoSpark and SpatialSpark. SpatialSpark and GeoSpark both provide some core functionality for dealing with spatial data like filter, joins and indexing. However, these frameworks both do not support spatio-temporal data and additionally, only GeoSpark

[5] http://tsusiatsoftware.net/jts/main.html
[6] https://github.com/jiayuasu/JTSplus

Tab. 1: Comparison of our STARK approach to GeoSpark and SpatialSpark

	GeoSpark	SpatialSpark	STARK
Language integrated DSL	x	x	✓
Support for Temporal Data	x	x	✓
Data Partitioning	✓	✓	✓
Indexing	✓	✓	✓
Persisted Indexes	x	✓	✓
Filter		no partitioning	
Contains	✓	(✓ - w/o Index)	✓
Intersects	✓	(✓ - w/ Index)	✓
WithinDistance	x	(✓ - w/o Index)	✓
Join	(✓ - pred. limitations)	(✓ - returns IDs)	✓
Nearest Neighbors	✓	x	✓
Clustering	x	x	✓
Skyline	x	x	(✓ - development)

supports nearest neighbors search as a more sophisticated data analysis operator. Both implementations do not provide a clustering or skyline implementation. Because of the respective design decisions and the resulting limitations of GeoSpark and SpatialSpark, we decided not to build our spatio-temporal engine on top of these platforms as this would have meant to rewrite a lot of code and rather built STARK from scratch. This way, we were able to fully integrate the DSL into Spark and build a flexible set of operators.

3 Requirements for Spatio Temporal Data Processing

Based on the application examples sketched in Sect. 1 and the current state of the art in spatio-temporal data processing we can identify several requirements for spatio-temporal Spark extensions:

native support for spatio-temporal data: As provided by spatial relational database systems and also defined in the SQL standard, spatial Spark should support appropriate data types in order to represent a set of standard geometries. Particularly, for the language-integrated APIs these data types should be mapped to native classes with similar APIs across the different language interfaces.

seamless integration into the Spark framework: In Spark, data processing is implemented as transformations of immutable distributed datasets (RDD, DataFrame) as well as actions returning data to the driver program. Together with the functional style of the Scala (or even Python and Java) API this composes a powerful and expressive way of formulating dataflow programs. Thus, a spatial/temporal extension should follow the same approach by supporting the RDD/DataFrame interface and defining spatial/temporal operations as transformations.

efficient and scalable processing of spatio-temporal data: Efficient processing of spatial data with non-trivial geometries requires appropriate spatial indexes such as the R-tree. The same holds for temporal data where temporal indexes such as interval trees are used. However, in Big Data analytics we cannot assume that data is always indexed in advance. Therefore, indexing should be optional as well as adaptive in the sense that indexes can be built on the fly before performing spatial/temporal operations and that such indexes are materialized for later reuse.

In order to exploit the benefits of a data-parallel platform like Spark, data partitioning is a crucial task. Particularly, spatial data requires space partitioning ensuring that closely located data objects are assigned to the same partition. Furthermore, the paritioning strategy should also handle data skew for a better load balancing, i.e. producing partitions with similar numbers of objects.

expressive spatial and temporal operations: Finally, a spatio-temporal framework should support a standard set of spatial and temporal predicates and query operators including filters, spatial and temporal joins, nearest neighbor search, and clustering. These operators should make use of indexes if available but also work on non-indexed data.

As described in Sect. 2, existing solutions do not fulfill these requirements completely yet. Thus, we have developed the STARK framework aiming at overcoming the deficiencies of current approaches.

4 The STARK Framework: Architecture & API

One of the main design goals of STARK is the seamless integration of spatio-temporal data and query operators into Apache Spark meaning that the user does not see any difference to the standard Spark API. For this purpose, we introduce new classes that add spatio-temporal operators to standard RDDs and encapsulate spatio-temporal data by a special class `STObject`. Fig. 1 gives an overview of the architecture of the STARK framework and its integration into the Spark ecosystem.

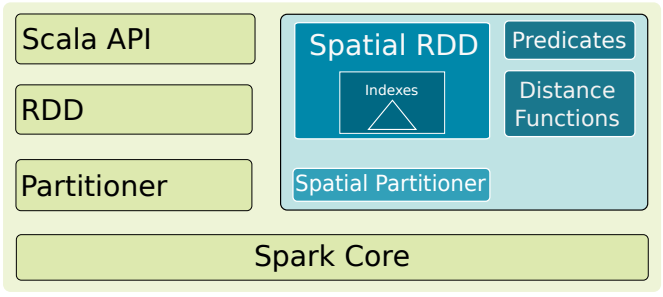


Fig. 1: Overview of STARK architecture and integration into Spark.

In the following, we describe the API of STARK comprising these classes as well as the RDD transformations for spatio-temporal operations. Furthermore, we present extensions to our dataflow compiler Piglet which extends the Pig Latin language by appropriate concepts.

4.1 DSL

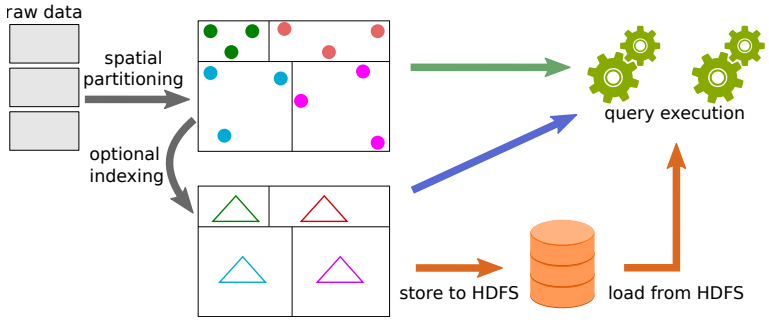


Fig. 2: Internal workflow for converting, partitioning, and querying spatio-temporal data

The goal of the STARK project is to create an easy to use DSL that can be used within any Spark program (written in Scala). This DSL should contain all necessary operations, as identified in Sect. 3 to comfortably work with spatio-temporal data. The possible workflow is shown in Fig. 2. An RDD is partitioned using a spatial partitioner and can optionally be indexed. Queries can be run on unindexed data as well as on indexed data. Indexes can be persisted and loaded again in other scripts.

As a basic data structure, STARK uses an `STObject` class. This class is used to represent the spatial and/or temporal component of any real world object. The class provides only two fields: (1) `geo` for storing the spatial component and (2) `time` to hold the temporal component of an object. To support spatial data without a temporal component, the `time` field may be left empty. Like many other Java based open source projects that deal with spatial data, STARK uses the JTS library with the JTSplus extension for internal representation of spatial objects and index structures.

The `STObject` class also provides various functions to test relations to other instances:

intersect(o) checks if the two instances (*this* and `o`) intersect in their spatial and/or temporal component,

contains(o) tests if *this* object completely contains `o` in their spatial and/or temporal component, and

containedBy(o) which is implemented as the reverse operation of *contains*

The above functions all consider both, the spatial and the temporal component of the objects. Thus, a check of the three above mentioned is only true, iff:

1. the check yields true for the spatial component, and
2. both temporal components are *not* defined or
3. both temporal components are defined and they also return true for the respective check.

Or, expressed more formally: for two objects o and p of type `STObject` and a predicate Φ:

$$\begin{aligned}\Phi(o,p) \;\Leftrightarrow\; & \Phi_s(s(o),s(p)) \wedge (\\ & (t(o) = \perp \wedge t(p) = \perp) \vee \\ & (t(o) \neq \perp \wedge t(p) \neq \perp \wedge \Phi_t(t(o),t(p))))\end{aligned}$$

Where $s(x)$ denotes the spatial component of x, $t(x)$ the temporal component of x, Φ_s and Φ_t denote predicates that check spatial or temporal objects, respectively, and $\perp$ stands for `undefined` or `null`.

Spark's core component are resilient distributed datasets (RDDs). An RDD is a generic in-memory partitioned collection that can be distributed among the cluster nodes. To add support for spatio-temporal operations into Spark, STARK provides `SpatialRDDFunction` classes that wrap a traditional RDD and implement the spatial operators. This follows the concept that Spark implements for special operators on Pair-RDDs, like a join. A join can only be executed if the input RDDs are Pair-RDDs, i.e. they contain 2-tuples `(k,v)`, where `k` is used as join attribute. Spark automatically creates a `PairRDDFunction` object with the input RDD as parameter, using Scala's implicit conversions. The `PairRDDFunction` class then implements the join method.

STARK also provides such implicit conversions that create a `SpatialRDDFunction` object of a Pair-RDD, where the key `k` is of type `STObject` [7]. The value `v` of that pair can be of any type and is maintained during all operations. The `SpatialRDDFunction` class implements all spatial operations supported by STARK: filtering, join, kNN, clustering, as well as indexing.

The implicit conversion is transparent to the users and creates a seamless integration into any Spark program. Users don't have to explicitly create an instance of any of STARKs classes (except `STObject`) to use the spatial operators.

We now show how to transform a normal RDD loaded from a text file into a `SpatialRDD` and how to use it: Consider an example where we have a dataset given as a CSV file that contains a list of events from various categories. The schema of that file might be: `(id: Int, description: String, category: String, wkt: String, time: Long)`. After loading, preprocessing, and transforming, we get an RDD of exactly that type: `RDD[(Int, String, String, String, Long)]`. We then create an `STObject` representing the location and time of occurrence from the WKT string and time field, respectively, of each entry:

```
val events = rdd.map { case (id, desc, category, wkt, time) =>
                    ( STObject(wkt, time), (id, desc, category) ) }
```

The resulting RDD is of type `RDD[(STObject, (Int, String, String))]`. We can now simply use this RDD to call the functions to filter with a predicate.

[7] In the following we will refer to such an `RDD[(STObject,V)]` as `SpatialRDD` and exclude the implicit conversion.

```
val qryTime = 1481287522
val qry = STObject("POLYGON((....))", qryTime)  // create a query object
val contain = events.containedBy(qry)  // events contained by the query region
val intersect = events.intersect(qry)  // events that intersect with the query
```

Unlike in GeoSpark, in STARK a single RDD can hold objects of different geometries. This means that an `STObject` in `events` can represent a simple point, but also a polygon or any other geometry that can be represented as WKT.

In addition to the filtering operators shown above, STARK has also built-in support for spatial joins, k nearest neighbor search, and clustering as well as for computing skylines, which is currently under development and not released yet. These operators will be explained in Sect. 6.

4.2 Piglet Integration

In addition to the language-integrated DSL for the Scala programming language we also provide support for our dataflow compiler Piglet [HS16] that implements an extended Pig Latin dialect. Piglet generates code not only for Apache Spark but also for Flink and the streaming variants Spark Streaming and Flink Streaming. In addition to the backend specific operators it offers a lot of extensions to support, e.g., Linked Data, Basic Graph Patterns (BGP), matrix data, R scripts, and embedded code.

The goal of Piglet[8] is to simplify the development of data processing programs. Scripts can be compiled, packaged, and executed on the local machine or a YARN/Mesos cluster with a single command avoiding the tedious tasks of compiling and deploying all dependencies to a cluster.

In order to add support for spatial data into Piglet, we have introduced a new Pig data type called `geometry` and added new operators for filter, join, and indexing. The `geometry` data type is mapped to the Scala class `STObject` and instances can be constructed, e.g., from a WKT string.

Spatial data processing is supported by two operators:

- In order to filter a bag using some spatial predicate the `SPATIAL_FILTER` operator can be used.
- A spatial join is performed using the `SPATIAL_JOIN` operator. Same as for the `SPATIAL_FILTER` the join predicate can be defined by the user.

In the following example we assume two data sets: The first contains the names of states and their respective border as a WKT string. The second data set contains a list of events given with their ID and the respective latitude and longitude coordinates. To find the countries that each event occurred in, we can join these two data sets using the `contains` predicate:

[8] https://github.com/dbis-ilm/piglet

```
countriesRaw = LOAD 'countries.csv' as (name: chararray, poly: chararray);
countries = FOREACH countriesRaw GENERATE name, geometry(poly) as cntry;

eventsRaw = LOAD 'events.csv' AS (id: chararray, lat: double, lon: double);
events  = FOREACH eventsRaw GENERATE id, geometry("POINT("+lat+" "+lon+")") as loc;

eventCountries = SPATIAL_JOIN countries, events ON contains(cntry, loc);
DUMP eventCountries;
```

Internally, Piglet represents the script as a dataflow plan – the logical representation of the operator graph. The plan is passed to a rewriter which applies optimization and transformation rules and then sends the rewritten plan to the code generator which uses a template file to create the source code for the selected target platform. When generating the code for the `SPATIAL_JOIN` in the example above, the code generator has to produce code to ensure the correct input/output format, i.e., in this example conversion from `(chararray, geometry)` to `(geometry, (chararray, geometry))` which is required to for the implicit conversion to the `SpatialRDDFunctions` object of which we can call the spatial functions.

The Piglet integration also supports indexing (see Sect. 5) which can be done by the `INDEX` operator. The operator allows to specify the parameters needed to partition and index the data:

```
eventsIdx = INDEX events ON loc USING RTree(maxCost=100, cellSize=1.0, order=5);
```

Here, an R-tree index is created after applying a cost-based partitioning (see next section), where `maxCost` defines the maximum cost per partition, `cellSize` the side length of a cell used for the partitioning, and `order` is the order of the tree, i.e., the number of entries per node. Using the index is transparent in the Pig script: users can simply apply a filter or join operation on the `eventsIdx` bag.

Currently, the Pig Latin extension for spatial data is available only for the Spark target platform as it uses the STARK library. Though, we are working on a Flink port of the STARK project with which we are able to activate this extension for the Flink target, too.

5 Partitioning and Indexing in STARK

5.1 Partitioning

Spark leverages the data parallelism of the Hadoop environment and a program is executed in parallel on different nodes, where each node processes a (small) portion of the complete dataset, called a partition. Spark uses built-in partitioners, e.g., a hash partitioner, to assign each data item to a partition. However, while for text data it might be enough to create partitions of equal size, for spatial data one would also like to exploit the locality of the spatial feature of the data items. Consider for example a hash partitioner and a spatial grid partitioner: while both partitioners may create partitions of equal size, i.e. the same number of data items in each partition, the partitions created by the hash partitioner contain

points from all over the data space disregarding their neighborhood, because the partition assignment is solely decided upon the hash value of the ID of the point, *h(x)*. Although we still can compute the result for, e.g., a spatial join operation, we cannot discard any partition apriori, because all partitions potentially contain join candidates. On the other hand, the spatial partitioner, in this case a simple spatial grid partitioner, divides the space using a grid and each resulting grid cell is a partition. If we now compute the minimum and maximum values for each dimension (e.g, latitude and longitude or x,y,z) we can easily decide if a partition contains potential join candidates or filter results, even without creating real indexes. We will talk about partition bounds for pruning at the end of this sub-section and describe how it is implemented in Sect. 6. Currently, STARK only uses the spatial component for partitioning (and indexing). However, in our current work we integrate the temporal component into both, partitioning and indexing.

Grid Partitioner. As briefly described before, the spatial grid partitioner creates partitions based on a grid over the data space. The partitioner accepts two parameters: the number of partitions in each dimensions (partitions per dimension, *ppd*) and the number of dimensions, where the latter has a default value of 2. To compute the cell size, the partitioner has to know the minimum and maximum values for each dimension. These values can be given as parameters or easily be computed by the partitioner in one single pass over the data. The side length in dimension i of a cell is determined by the minimum and maximum values for this dimension and the *ppd*:

$$length_i = \mid max_i - min_i \mid / ppd$$

To compute the partition a given point p belongs to, the partitioner has to simply calculate the partition ID $partitionId$. For a two dimensional scenario the formula would be:

$$x = \lfloor \lfloor \mid p_1 - min_1 \mid \rfloor / length_1 \rfloor$$

$$y = \lfloor \lfloor \mid p_2 - min_2 \mid \rfloor / length_2 \rfloor$$

$$partitionId = y * ppd + x$$

Binary Space Partitioner. The disadvantage of the naïve Grid Partitioner is that if the data points are skewed with only a few outliers throughout a large data space, the partitioning will result in lots of partitions were many of them are empty, some containing only very few data points, maybe only one or two, and only a small number of partitions with almost all data points. In an environment like Spark, this means that one executor has to perform all the work, i.e. process all data points, while other executors that were assigned an empty partition have no work to do. Thus, we additionally implemented the Binary Space Partitioner presented by He et al. in [H+14]. The advantage of this partitioning method is that one can define a maximum cost for a partition. First, the data space is divided into small quadratic cells of a given side length. Then, all possible partitioning candidates along the cell bounds in all dimensions are evaluated. After testing all possible partitioning candidates, the partitioning with smallest cost different between both candidate partitions is applied.

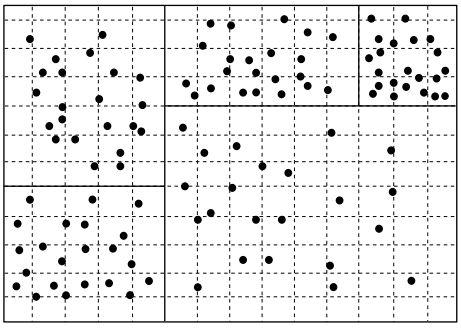


Fig. 3: A Binary Space Partitioning resulting in 5 partitions for a maximum cost of 22. Dashed lines represent cells and solid lines mark boundaries for generated partitions.

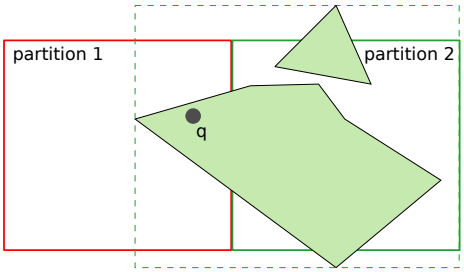


Fig. 4: Two neighbor partitions (solid lines) and the extent (dashed line) for partition 2 based on the contained geometries.

This results in two partitions for which the process is repeated recursively, if the according partition is longer than one cell length in at least one dimension and its cost is greater than the given maximum cost. While the authors of [H+14] introduce a sophisticated cost formula that also takes disk accesses and loading times into account, we simply consider the number of data points inside a partition as its cost. Applying this partitioning approach will result in partitions of almost equal cost, if the cell size is chosen reasonably according to the data. Fig. 3 shows a sample partitioning where the space is divided into 14×12 cells and results in 5 partitions with a maximum cost of 22.

Partitioning Polygons. The spatial partitioners decide to which partition a geometry belongs based on its position in space. Both introduced partitioners are based on grid cells of a fixed size, i.e. a width and height in two dimensional space. For points, the partitioners simply check which grid cell contains the current point and use this information to assign it to a partition. If the RDD contains not only points, but also polygons, these polygons may be larger than a grid cell. To decide to which partition a polygon belongs, the partitioners use the centroid point of that polygon.

While we can assign polygons to cells and therefore to partitions, this would create incorrect results when we use partitioning information to prune partitions before executing a join or filter operation: We might prune a partition based on its computed dimensions, the contained polygons, however, might span beyond the partition bounds and actually match the filter or join predicate. Fig. 4 shows an example of two neighbor partitions and a query point $\mathtt{q}$, that lies within partition 1. If we wanted to find all geometries that contain $\mathtt{q}$, we would prune partition 2 and falsely not find polygon $\mathtt{p}_1$ (which is assigned to partition 2) as result. Therefore, in addition to the computed bounds, we also keep the *extent* of a cell/partition. This extent information is stored as a rectangle (in 2D space) and is created from the minimum and maximum x and y values of the cell/partition bounds and of all elements inside that cell/partition. For partition pruning, we then consider not the theoretical partition bounds, but the partition's extent. For the BSP, the extent of a partition is computed from all extents of the cells inside that partition.

5.2 Indexing

Repartitioning the data according to its spatial distribution can help to improve query performance. However, since all data items within a partition are candidates for the current spatial predicate of a query, nothing can be omitted and they all have to be evaluated. To further improve performance, STARK additionally can create an index for each partition. Theoretically, STARK can index a partition using any in-memory spatial index structure implementation. Currently, we support the R-tree index structure provided by the JTS library and plan to include more index structures in upcoming versions. In their Spark program, one can choose between the following three indexing modes:

No Index No index structure is used, and all data items are evaluated for query processing. This can be useful if the costs for creating and querying an index exceed the costs for processing all items (e.g., a full table scan). Note, in this case it does not matter how the RDD is partitioned.

Live Indexing During live indexing, the data is repartitioned using a given partitioner, if it was not partitioned before. Upon evaluation of a spatial or spatio-temporal predicate, all data items of a partition are first put into an index structure. Then this index is queried according to the spatial predicate of the query and, after pruning the candidates from the tree query, the overall result is returned.

Persistent Index For persistent indexing, the content of a partition is put into an index structure which is then used to evaluate the actual predicate. In contrast to live indexing, this execution mode changes the type of the input RDD from `RDD[(STObject , V)]` to `RDD[RTree[STObject , (STObject ,V)]]`. This means that the resulting RDD consists of R-tree objects instead of single tuples. This way, the index can be reused for subsequent operations. Furthermore, it can even be materialized to disk and loaded again for another execution. Thus, the index can be shared among different scripts to avoid the costly creation over and over again. Fig. 5 shows a data space which is partitioned into five partitions with equal costs. These partitions are indexed using the persistent index mode which results in an RDD with five entries.

As stated before, in our current work we investigate how to integrate the temporal component into indexing. One solution is to simply create two index structures, one for spatial and for temporal data, query both, and finally merge the results. However, in the literature there are dedicated approaches for spatio-temporal indexing ([MGA03; TVS96; ZSA99]), that we currently evaluate and will finally integrate into STARK.

6 Operator Implementation

Filter. The spatial filter operator for the different predicates is implemented as extra methods in the `SpatialRDDFunction` classes. An RDD can be filtered using the *contains*, *containedBy*, *intersects*, and *withinDistance* predicates. The first three mentioned predicates expect exactly one argument: the reference object (of type `STObject`) for which the respective predicate is to be evaluated. The operation is implemented as a Spark transformation and hence does not have any shuffling cost as it can be evaluated locally on a node's partition.

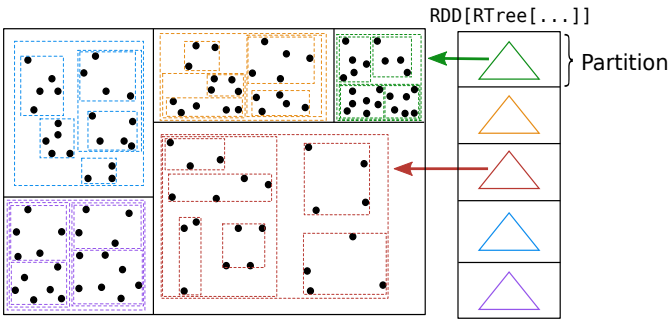


Fig. 5: A data space partitioned into five partitions which are indexed using an R-tree.

Algorithm 1 Filter operator and partition pruning - without indexing.

```
1:  procedure INTERSECTS(partition, qryObj)
2:      checkPart ← true
3:      if rdd.partitioner isA SpatialPartitioner then      ▷ Pruning only for spatial partitioned RDDs
4:          sp ← rdd.partitioner as SpatialPartitioner
5:          extent ← sp.partitionExtent(partition)
6:          checkPart ← extent.intersects(qryObj)    ▷ Check only if qryObj intersects with partition
7:      end if
8:      if checkPart then                                   ▷ Partition can contain results
9:          for each g in partition.iterator do
10:             if qry.intersects(g)) then                  ▷ Apply predicate
11:                 emit g
12:             end if
13:         end for
14:     end if
15: end procedure
```

The *withinDistance* predicate expects not only the reference object to which the distance should be computed, but also the distance function to use. The idea is to suppoert data of a cartesian as well as, e.g., a geodedic coordinate system which require different distance metrics for accurate computations.

The filter operators can prune partitions that cannot contain result tuples, if the RDD was partitioned using a spatial partitioner. Algorithm 1 shows the pseudocode of the *intersects* filter operator without indexing for a single partition. Spark executes the code for each partition in parallel without any additional programming effort. If the RDD was not partitioned using a spatial partitioner, we have to test each element in the partition with the predicate function. For the actual filtering, we iterate over all elements of that partition and apply the respective predicate function.

Nearest Neighbors. The nearest neighbor operator follows a straightforward implementation. Since the operators are executed in parallel without the nodes communicating with each other, an executor cannot decide alone, if its assigned partition contains elements that belong to the result. Thus, we compute the k nearest neighbors for each partition individually by computing the distance of each element to the query object, sort these elements by their respective distance to the query object in ascending order, and then return only the first k items. This results in n lists of k elements, if the RDD consists of n partitions. We then apply a global sort of these $n \times k$ elements and return only the first k items as the final result.

Algorithm 2 Join operator for contains predicate - with live indexing.

```
1:  procedure GetPartitions(leftRDD, rightRDD)
2:      for each l in leftRDD.partitions do          ▷ Check for spatial partitioner left out for brevity
3:          for each r in rightRDD.partitions do
4:              if l.intersects(r) then              ▷ Partitions must intersect to contain join results
5:                  emit new JoinPartition(l, r)
6:              end if
7:          end for
8:      end for
9:  end procedure

10: procedure LiveIndexJoin(joinPartition, predicateFct)
11:     tree ← new RTree()
12:     for each l in joinPartition.leftIterator do          ▷ Build index with all items of leftRDD
13:         tree.insert(l)
14:     end for
15:     for each r in joinPartition.rightIterator do
16:         candidates ← tree.query(r)                       ▷ Query index with each entry in rightRDD
17:         for each c in candidates do                      ▷ R-tree returns candidates only
18:             if predicateFct(c,r) then                    ▷ Apply predicate
19:                 emit (c, r)                              ▷ c is from leftRDD
20:             end if
21:         end for
22:     end for
23: end procedure
```

Join. A spatial join is a join operation using a spatial predicate like *contains* or *intersects*. In Spark however, only equi-joins are supported and θ-joins have to be implemented by the user. Hence, STARK provides its own implementation of join algorithms.

The join function accepts the other spatial RDD to join with as well as the join predicate. The predicate can be given as a function or as a identifier (an instance of a enumeration). Algorithm 2 shows the implementation of a join operator with partition pruning. To compute a join in Spark, we first have to produce the cartesian product of the partitions of both RDDs. When generating this cartesian product, we can check if the two partitions match the join predicate, e.g., intersect each other. If they do not match this predicate, this combination will not contain join partners and we can safely omit this combination. For combinations that can contain join partners, we create a new instance of a `JoinPartition` that contains references to these two partitions of the respective input RDDs.

For the actual computation of the join, the Spark framework will provide an executor with a partition which is now of type `JoinPartition`. For live indexing, we now iterate over the elements of the left RDD and insert the elements into an R-tree. Then, we query the R-tree with all elements of the right RDD and apply the predicate function to find join partners. Note, that the predicate may test the spatial or spatio-temporal components of the elements (same for the filter operator).

Clustering Another important operator for spatio-temporal data is clustering to identify groups of objects that occur close to each other in space and/or time. STARK comes with its own implementation of the density-based clustering algorithm DBSCAN for Spark. Our

implementation is inspired by the MR-DBSCAN algorithm for MapReduce [H+14] and exploits Spark's data parallelism as well as makes use of the spatial partitioners.

The main idea of this algorithm is as follows: first, the data is partitioned in a way that all partitions are equally-sized for a better load balancing. Next, these partitions are extended in each direction by the value of the ϵ parameter of DBSCAN to overlap with their neighboring partitions. This partitioning step requires a shuffling of all data. Note, that some objects (which are contained in the overlap regions) are assigned to multiple partitions. Then, for each partition a local DBSCAN is performed in parallel to identify partition-local clusterings. Because this step produces different clusterings for each partition, an additional merge step is needed where objects from overlap regions assigned to multiple clusters are used to merge these clusters by constructing a graph of cluster pairs. In this graph, nodes represent local clusters and edges denote inter-partition relationships between clusters which can be merged. Based on this information, the objects are assigned to a single cluster in the final step. A crucial step in this algorithm is the partitioning: particularly, for skewed data a simple hash-based or even grid-based partitioning will result in an imbalance of numbers of objects per partition and, therefore, very different efforts for computing the local clustering. In order to avoid this problem, our DBSCAN implementation uses the spatial partitioner introduced in Sect. 5 to determine the initial partitions.

7 Evaluation

In this evaluation we will examine the performance of STARK and compare it to the competitors GeoSpark and SpatialSpark in their latest versions found on GitHub (GeoSpark: v0.3, SpatialSpark: v1.1.0). For our experiments, we used a cluster of 16 nodes where each node has an Intel Core i5 processor, 16GB RAM, and a 1TB disk. All experiments were executed in YARN mode with 16 executors and 2 cores per executor. On our cluster we run Ubuntu 14.04 with Hadoop 2.7, Spark 1.6.2 (because at the time of writing GeoSpark only supported this Spark version), Scala 2.11, and Java 1.8. Every test case was executed five times and the best result for each platform was chosen for comparison.

To create data sets, we had to comply with the requirements of the other two frameworks: GeoSpark can only work with data that contain only points or polygons, but not mixed data. SpatialSpark only supports RDDs with a Long value at the first position and the geometry at the second position. Other values are not allowed in addition to that. Since the other frameworks do not support spatio-temporal data, we only test spatial predicates.

We chose a sample of the `taxi` trip data[9] with trip information like pick up/drop off times and locations that represent our points-only data set and it contains 34 million entries. Additionally, we used a data set with only polygons that is created from an Open Street Map `world` dump. From this dump we exported the polygons of all borders, except of national borders, resulting in 322000 polygons. For our experiments with the join operator, we used a third data set called `blocks`[10] that contains around 38000 polygons representing all census

[9] http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
[10] http://www1.nyc.gov/site/planning/data-maps/open-data.page

tracts and blocks of New York City. The *taxi* and `blocks` data sets were used in [YZG15], too. In addition to the results shown in this paper, we publish more results with other larger data sets on GitHub[11].

First, we conducted a test set for filter operations. In these experiments we created all combinations of supported partitioners with both live indexing and no index mode. Where applicable, all partitioners are executed with the same parameters. There are three filter operators that (1) find all polygons in `world` that *contain* a given query point, (2) find all polygons in `world` which *intersect* with a given query polygon, and (3) find all points in `taxi` that are *within a maximum distance* to a given point.

During the test construction some limitations and issues of the platforms came to light. In GeoSpark for example, one cannot combine spatial partitioning and indexing. Furthermore, the *contains* predicate was implemented the wrong way around and range filters with index only returned the candidates of an R-tree query without applying a predicate. We had to fix this to achieve comparability. For SpatialSpark, partitioning is not possible at all for range queries. On top of that, without index only *contains* and *withinDistance* and with persistent index only *intersects* is possible. The results of these filter operators for the best partitioner of each platform are shown in Fig. 6.

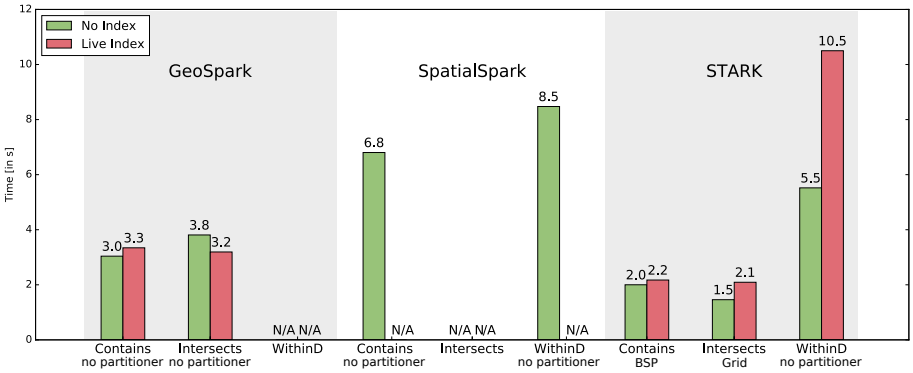


Fig. 6: Runtimes for filter operators with different indexing modes

Regarding the feature width, it can be seen that STARK offers the best use case coverage supporting all three operators both with and without using an index. STARK performs better for all executed filters than its competitors. Even for our relatively small test data set, the differences are clear and we expect them to be even greater for larger data sets. Especially for the *contains* and *intersects* queries partition pruning pays off. Due to internal implementation and design decisions, partition pruning cannot be applied with the *withinDistance* predicate and it will be addressed in future work. Different than expected the live index does not bring any benefit and is almost always slower in comparison to no index usage. Since the selectivity is quite high (there are less than 5 result tuples in each operation), the reason for this is probably the small size of the data and building the index requires more time than the faster lookup can improve the query. For larger data sets and larger partitions, the indexing will surely be much faster.

[11] https://github.com/dbis-ilm/spatialbm

For filtering, GeoSpark performs best with no partitioning at all. SpatialSpark has no partitioning possibility for filter queries and, thus, the results for no partitioner is presented here. In the case of STARK, for each query another partitioning mode performs best. The reason that without spatial partitioning GeoSpark performs best is that the `taxi` data is highly clustered with only a few outliers. From the 34 million trip points, only a few hundred points are outside of Manhattan and some of them are very far away. Hence, millions of points are only a few centimeters away from each other. To achieve a good partitioning, a grid or BSP partitioner has to be configured with, e.g., a large number of very small cells to distribute those dense points to different partitions and hence create a balanced workload on the executors. With the highly clustered data, the partitioners tend to create a small number of partitions with a large number (millions) of points causing this executor to have all the work to do. Spark's hash partitioner, however, creates an even distribution, and thus, GeoSpark performs better without spatial partitioning. STARK however, was able to find a good spatial partitioning and therefore was able to apply partition pruning for additional speedup.

After examining the filter operators, we conducted a set of experiments for join operators. Since the `taxi` and `world` data sets contain a large number of entries each, we had to reduce the number of points and polygons in the respective input data sets to account for the limited hardware and to achieve reasonable execution times for repeatable experiments. We reduced the number of points by taking a sample of the original `taxi` data set, which contains 1328 points.

Also the implementation of the join tests revealed some limitations of the competitors. SpatialSpark for instance enforces the usage of an index when no partitioning is used. Outputting the result as ID tuple is also very limiting as already mentioned in Sect. 2. For GeoSpark partitioning is required for every join variant. Furthermore, when not using an index the predicate *contains* is automatically used while with indexing only the unfiltered candidates of the R-tree query are returned, which may be not the correct result. Although the *withinDistance* predicate is possible, it is limited to point geometries exclusively.

The join operations were also executed for all combinations of partitioners and indexing modes and the results for the join with *contains* predicate is shown in Fig. 7. We see that

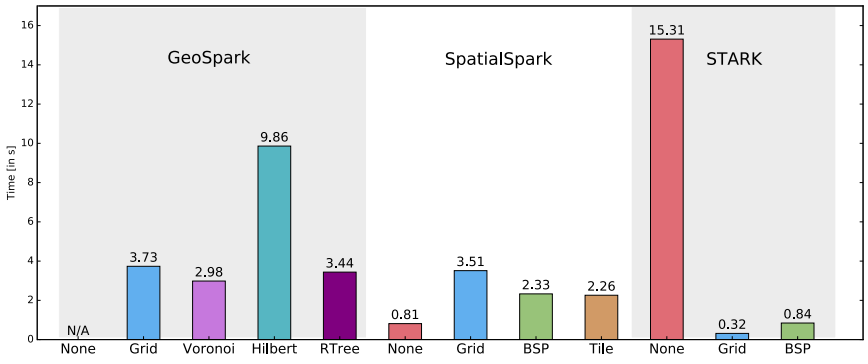


Fig. 7: Runtimes of the join operator with *contains* predicate for each partitioner using live index.

STARK outperforms GeoSpark and SpatialSpark if a spatial partitioner is used. Without spatial partitioning, we have to produce an expensive cartesian product and check all partition combinations for possible join partners. Without a spatial partitioner, SpatialSpark collects all entries of the right relation on one node and builds an R-tree. This tree is made available to all nodes using Spark's broadcast variable. This is obviously a good approach as long as data fits in memory. The runtime differences come from the time needed for partitioning and probably again from STARK's partition pruning ability.

Lastly, we ran an experiment with STARK's persistent indexing. Tab. 2 shows the runtimes for live and persistent indexes for a join. This join has to find all elements from `world` that contain points from an `event` data set which has 15000 events from all over the globe. This basically shows the overhead of creating the index. For persistent index, the index is already present to the operator which can use it for its operation. Also, a subsequent operator can make use of this index without recreating it.

Tab. 2: Comparison of live & persistent indexing

	Live Index	**Persistent Index**
BSP	12.6 sec	8.3 sec
Grid	20.6 sec	18.5 sec

8 Summary

In this paper we presented our STARK framework for analyzing large spatio-temporal data sets on Apache Spark. STARK integrates seamlessly into a Spark program by providing automatic conversion methods that, from a user perspective, add operators with spatial and spatio-temporal predicates to Spark's RDDs. STARK provides spatial partitioners and different indexing modes. A generated index can be stored persistently and loaded again by other scripts. Operators support these indexes, but also work on unindexed data. For further speedup, the operators can benefit from the spatial partitioning by not processing partitions that cannot contain any result tuples. In our experimental evaluation we compared STARK to GeoSpark and SpatialSpark and showed that it performs better and also provides a much more complete, flexible, and well integrated set of operators. In addition to the filter and join operators, STARK provides a DBSCAN clustering operator. In our ongoing work we extend the framework by more analysis operators like Skyline or Complex Event Patterns. For the skyline operator an angular partitioner, as described in [CHW12], is being implemented. Furthermore, we will integrate the temporal aspect more deeply into the framework and will use the temporal data for partitioning and indexing, too.

Acknowledgements This work was partially funded by the German Research Foundation (DFG) under grant no. SA782/22.

References

[AW+13] Aji, A.; Wang, F., et al.: Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce. VLDB Endow. 6/11, pp. 1009–1020, Aug. 2013.

[CHW12] Chen, L.; Hwang, K.; Wu, J.: MapReduce Skyline Query Processing with a New Angular Partitioning Approach. In. IPDPS, pp. 2262–2270, May 2012.

[EM13] Eldawy, A.; Mokbel, M. F.: A demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data. In. VLDB Endow., Aug. 2013.

[EM15] Eldawy, A.; Mokbel, M. F.: SpatialHadoop: A MapReduce Framework for Spatial Data. In: ICDE. Seoul, 2015.

[F+13] Fox, A.; Eichelberger, C., et al.: Spatio-temporal indexing in non-relational distributed databases. In: Big Data. 2013.

[H+14] He, Y.; Tan, H., et al.: MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. FCS 8/1, pp. 83–99, 2014.

[HS16] Hagedorn, S.; Sattler, K.-U.: Piglet: Interactive and Platform Transparent Analytics for RDF & Dynamic Data. In. WWW 2016 Companion, 2016.

[IBM13] IBM: IBM DB2 10.5 Spatial Extender User's Guide and Reference, July 2013.

[MGA03] Mokbel, M. F.; Ghanem, T. M.; Aref, W. G.: Spatio-temporal access methods. IEEE Data Eng. Bull. 26/2, pp. 40–49, 2003.

[Na] National Geospatial-Intelligence Agency, N.: GeoWave, http://ngageoint.github.io/geowave, Accessed Dec. 13, 2016.

[Or14] Oracle: Oracle Database 12c: An Introduction to Oracle's Location Technologies, http://download.oracle.com/otndocs/products/spatial/pdf/12c/oraspatialandgraph_12c_wp_intro_to_location_technologies.pdf, Sept. 2014.

[TVS96] Theodoridis, Y.; Vazirgiannis, M.; Sellis, T.: Spatio-temporal indexing for large multimedia applications. In: ICMCS. Pp. 441–448, 1996.

[WP+14] Whitman, R. T.; Park, M. B., et al.: Spatial Indexing and Analytics on Hadoop. In. SIGSPATIAL, 2014.

[YWS15] Yu, J.; Wu, J.; Sarwat, M.: Geospark: A cluster computing framework for processing large-scale spatial data. In: International Conference on Advances in Geographic Information Systems. SIGSPATIAL, 2015.

[YWS16] Yu, J.; Wu, J.; Sarwat, M.: A demonstration of GeoSpark: A cluster computing framework for processing big spatial data./, pp. 1410–1413, 2016.

[YZG] You, S.; Zhang, J.; Gruenwald, L.: Large-scale spatial join query processing in cloud, https://github.com/syoummer/SpatialSpark, Accessed Sept. 13, 2016.

[YZG15] You, S.; Zhang, J.; Gruenwald, L.: Large-scale spatial join query processing in cloud. In. ICDEW, pp. 34–41, 2015.

[ZSA99] Zimbrão, G.; de Souza, J. M.; de Almeida, V. T.: The temporal R-tree, tech. rep., Technical Report ES492/99, COPPE/Federal University of Rio de Janeiro, Brazil, 1999.

Data Integration

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 145

IncMap: A Journey towards Ontology-based Data Integration

Christoph Pinkel[1] Carsten Binnig[2] Ernesto Jimenez-Ruiz[3] Evgeny Kharlamov[3]
Andriy Nikolov[1] Andreas Schwarte[1] Christian Heupel[1] Tim Kraska [2]

Abstract: Ontology-based data integration (OBDI) allows users to federate over heterogeneous data sources using a semantic rich conceptual data model. An important challenge in ODBI is the curation of mappings between the data sources and the global ontology. In the last years, we have built *IncMap*, a system to semi-automatically create mappings between relational data sources and a global ontology. *IncMap* has since been put into practice, both for academic and in industrial applications. Based on the experience of the last years, we have extended the original version of *IncMap* in several dimensions to enhance the mapping quality: (1) *IncMap* can detect and leverage semantic-rich patterns in the relational data sources such as inheritance for the mapping creation. (2) *IncMap* is able to leverage reasoning rules in the ontology to overcome structural differences from the relational data sources. (3) *IncMap* now includes a fully automatic mode that is often necessary to bootstrap mappings for a new data source. Our experimental evaluation shows that the new version of *IncMap* outperforms its previous version as well as other state-of-the-art systems.

1 Introduction

As large volumes of data are being constantly created in a variety of domains, the challenge of data integration becomes more and more important to get a holistic view on existing knowledge. Example use cases include web data analysis, the reconciliation of enterprise internal data, as well as applications in science or medicine. Integrating such data into a common model allows correlating information and hence discovering new and interesting patterns [Bh15, DS13].

One recent approach to this problem is ontology-based data integration (OBDI), where data sources are integrated via a global ontology. The advantage of using an ontology as a global view is its semantic richness, allowing domain experts to model their information needs in a conceptual high-level model. However, data in many of today's applications is still commonly stored in relational databases. To achieve an integration of these data sources, the relational database schemata of the sources have to be mapped into the unified global ontology (called *RDB2RDF* mappings further on).

Creating such *RDB2RDF* mappings manually is a time consuming task, which requires considerable effort and expertise. To minimize the required costs, several different systems (e.g., [Ji15, Pi13, Kn12, TSM13, Au05]) have recently been proposed that assist users in

[1] fluid Operations AG, Walldorf, Germany
[2] Brown University, Providence, RI, USA
[3] University of Oxford, Oxford, UK

creating these mappings in a semi-automated or even fully automated way. To provide such assistance, we have previously built semi-automatic system called *IncMap* [Pi13], which we have put to use in practice over the last few years in different application domains.

Contributions: The mapping generation in the first version of *IncMap* [Pi13] was mainly based on lexical as well as structural similarities between the schema elements of the relational data source and the target ontology. For example, if an entity *Author* has a relation to an entity *Paper* there should exist similar entities in both the ontology and the relational schema as well as a path that links both entities.

However, based on the use of *IncMap* over the last years we have seen that *IncMap* was not able to find more complex mappings in many real-world scenarios which had two main reasons: First, ontologies follow an open-world approach (i.e., not everything must be explicitly modeled if it can be derived by reasoning) whereas relational schemata follow a closed-world approach. Second, many of the high-level concepts such as inheritance can not be directly modeled in a relational schema and are often implemented using different modeling patterns.

Based on this experience, we have developed a new version of *IncMap* that tackles these problems. The main features of the new version that provided most benefits are: (1) As a first major extension, *IncMap* is now able to leverage knowledge derived from reasoning over the input ontology, and (2) at the same time utilizes information about typical design patterns in relational databases. To the best of our knowledge, *IncMap* is the only direct system that combines these two approaches into a mapping system. Both these extensions have shown to be extremely fruitful and are the main reason why *IncMap* currently outperforms other state-of-the-art systems. (3) In comparison with its predecessor, *IncMap* now also supports fully automatic mapping generation to bootstrap the mapping for larger data sources. Moreover, we have also added minor extensions such as a better lexical matching and a number of engineering improvements.

The contributions of this paper are three-fold: (1) We present all these new features of *IncMap* in detail. (2) In order to show the effectiveness of all our extensions, we compare *IncMap* against other state-of-the-art systems using our recent benchmark suite for benchmarking ODBI integration systems, called RODI [Pi15].[4] Our results show that *IncMap* improves the quality of the generated mappings significantly over its predecessor and typically outperforms other state-of-the-art systems. (3) In addition, we also present the results of a user-study. We have built the study on a small, real-world industry mapping problem to demonstrate the utility of *IncMap* in practice, independent from any synthetically designed benchmark.

Outline: The remainder of this paper is organized as follows. First, in Section 2 we start with an overview of our extended system *IncMap* that leverages the novel *IncGraph+* model to support the before-mentioned extensions. In Section 3, we discuss in detail

[4] `https://github.com/chrpin/rodi`

the construction of the *IncGraph+* model and then elaborate on the role of reasoning in Section 3.2 and the use typical design patterns in Section 3.3. Afterwards, in Section 4 we explain how the this information is leveraged to generate mappings from the extended *IncGraph+* model. Finally, we present our experimental evaluation in Section 5 and discuss the results of our user study in Section 6. We conclude with related work in Section 7 and a summary in Section 8.

2 *IncMap* Overview

IncMap generates *RDB2RDF* mappings between a given relational data source and a target ontology based on the following procedure that operates in five steps: (1) creating source and target *schema graphs* from the relational schema and the ontology, (2) use reasoning and heuristic pattern *annotation* to infer additional information, (3) apply initial lexical matching to build a *matching score* between relation schema and the ontology, (4) refine matching scores using a *fixpoint computation*, and (5) generate *RDB2RDF mappings*.

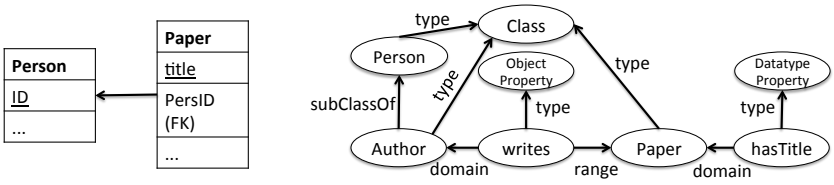
This procedure can be repeated after the user has verified or rejected individual mappings of the previous iteration. This allows *IncMap* to integrate user feedback and create mappings in an incremental manner.

In the following, we give an overview of all these steps. Details and a more formal explanation are given afterwards in Section 3 and Section 4.

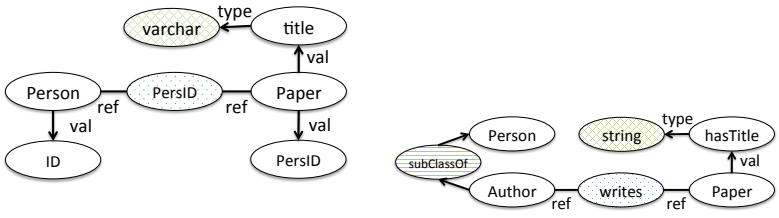
1. Creating Schema Graphs: As a first step, the source and target schema graphs are created using our *IncGraph+* model. In order to construct the source and target schema, we iterate over all schema elements in the relational schemata and the ontology and create the two graphs.

Figure 1 depicts a small example of the schema graph construction from the conference domain. The relational schema (Figure 1a) and the ontology (Figure 1b) both capture the same information about persons and papers. In a first step, the relational schema and the ontology are mapped into the source and target schema graph as shown in Figure 1c and Figure 1d that consists of typed nodes and labeled edges. A formal definition of the schema graph will be given in the following section.

While we can see already some clear correspondences between the elements in both schema graphs that result from lexical and structural similarities (e.g., *Paper*s and their *title*s), some other schema elements are harder to match. For example, while *Author* is a dedicated concept in the ontology it does not have a direct correspondence in the relational schema. The reason is that in the relational schema, *Author*s are modeled as *Person*s who have authored at least one paper. This results from a typical modeling pattern of how inheritance is implemented in a relational schema.



(a) Relational Schema $\mathcal{R}$ (b) Ontology $\mathcal{O}$



(c) Basic *IncGraph+* ($\mathcal{R}$) (d) Basic *IncGraph+* ($\mathcal{O}$)

Fig. 1: Example: schema and ontology, with basic *IncGraph+* representations.

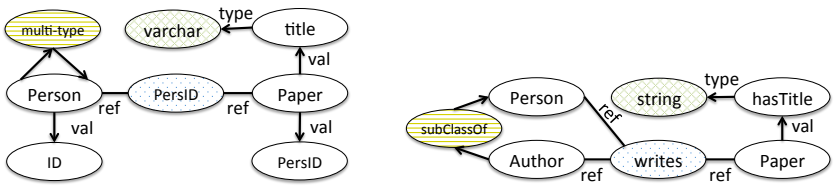
Moreover, there are many more subtle differences: e.g., while on the database side foreign keys (i.e., referential constraints) model a clear direction and allow to deduce cardinality constraints, object properties in an ontology are do not yield any cardinality information.

In order to simplify the subsequent matching step (3), we also annotate each node in a schema graph with a type indicated by the color coding in both figures. For example, the green-striped nodes in a schema graph represent the *data type* information. We explain the different types that we support in *IncMap* later. Important is that for the matching step (3), only nodes with the same color need to be considered as potential correspondences.

2. Reasoning and Patterns: As discussed before, there exist several possible distortions between the schema graph representations of a relational schemata and an ontology. These distortions have two main causes: First, ontologies typically follow an open-world approach (i.e., not everything must be explicitly modeled if it can be derived by reasoning) whereas relational schemata follow a closed-world approach. Second, many of the high-level concepts such as inheritance can not be directly modeled in a relational schema and are often implemented using different modeling patterns.

These problems also materialize in Figure 1. Intuitively, the *Person* class is the most accurate match for the *Person* table while the *Author* class has most probably no match candidate. This is because the *subClassOf* connection between *Author* and *Person* is not explicit anymore neither in the schema graph of the relational schema (since it was never modeled) nor the schema graph of the ontology.

For this reason, as a second step we apply reasoning techniques on the input ontology and use heuristics to annotate patterns on the source database. Figure 2 depicts derived knowledge from reasoning and annotated patterns for the example in Figure 1.



(a) Full *IncGraph*+ ($\mathcal{R}$) (b) Full *IncGraph*+ ($\mathcal{O}$)

Fig. 2: Example (ctd.): advanced *IncGraph*+s.

In Figure 2a, the relational schema graph now annotates the *Person* node with a pattern, which *heuristically* states that this table is very likely to contain *individuals* of several types (e.g., using sub classes or sibling classes). This information can now be used to derive a new correspondences between *Author* node in the schema graph of the ontology and the *Person* node in the schema graph of the relational data source.

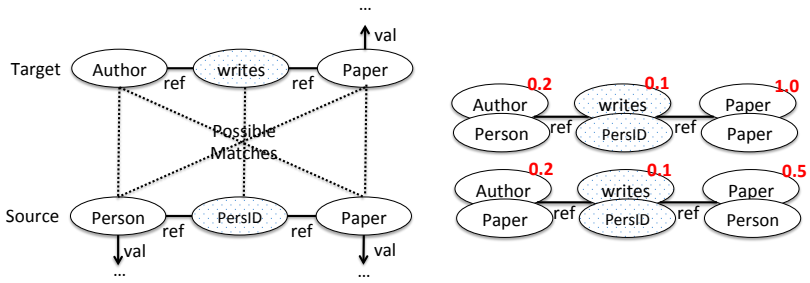
Additionally, a new reference edge is added to the ontology schema graph which directly connects *Person* and *Writes*. This knowledge is derived through reasoning and basically states that some persons in the ontology write papers. In our example this encourages correspondences between the *Person* nodes in both schema graphs since they are now not only lexically similar but also structurally.

3. Matching Step: Based on a source and target schema graph, we next calculate the initial matching. Figure 3 illustrates a simplified initial matching for the two schema graphs of the previous example.

For creating the initial matching, each node in the source graph is paired with each node in the target graph that has the same color (or type) into possible matches. Figure 3a shows the possible pairs for an excerpt of the schema graphs in Figure 2. Afterwards, only those pairs in the set of possible matches are kept that have the same set of labeled edges (in the source and target graph).

Based on the remaining possible matches a so called pairwise connectivity graph (PCG) is created where each subgraph represents a possible alignment of nodes in the source and target schema graph. Figure 3b shows the PCG for our example where the upper node in each subgraph is always from the target schema graph (i.e., the ontology) and the lower node from the source schema graph (i.e., the relational schema). For each pair of nodes in the PCG an initial score is calculated using a lexical matcher (e.g., using Jaccard similarity).

4. Fixpoint Computation: In subsequent next steps, *IncMap* then computes a fixpoint computation using the PCG. The fixpoint computation serves to refine match scores based



(a) Source and Target *IncGraph+*s (b) Pairwise Connectivity Graph

Fig. 3: Matching *IncGraph+*s (Simplified Excerpt)

on the graph structure. The fixpoint computation in *IncMap* follows the idea of the similarity flooding algorithm described in [MGMR02]. Intuitively, the process favors nodes in larger subgraphs over smaller ones and increases the scores of strongly connected nodes.

Different from the original similarity flooding algorithms, however, we introduce modifications for features such as weighted edges and selectively activating edges during the fixpoint computation (e.g., based on the annotated patterns). Moreover, in order to additionally support incremental mapping scenarios where user feedback is available, we can accommodate information on partial mappings. However, *IncMap* is also able to generate mappings completely without any user feedback.

5. Mapping Generation: Finally, mappings are generated from the correspondences resulting from the PCG. Correspondences are selected based on the highest matching scores of the fixpoint computation to form a consistent alignment interpretation; i.e., each node in the original target schema graph has maximally one correspondence in the source schema graph. To enable the mapping generation, we encode provenance information with all nodes of the PCG to refer back the nodes in the source and target schema graphs.

3 *IncGraph+*

In this section, we formally define the schema graphs that we create as a first step of *IncMap*. The data model used for a schema graph in our recent version of *IncMap* is called *IncGraph+*.

3.1 Basic *IncGraph+*

An *IncGraph+* is defined as a labeled graph $\mathcal{G} = (\mathcal{V}, \texttt{Lbl}_\mathcal{V}, \mathcal{E}, \texttt{Lbl}_\mathcal{E}, \mathcal{W}_\mathcal{E})$. It can be used as input by the matching algorithm of *IncMap*. $\mathcal{V}$ is a set of vertices, $\mathcal{E} = \mathcal{E}_d \cup \mathcal{E}_u$ is a set of directed $\mathcal{E}_d$ and undirected $\mathcal{E}_u$ edges, $\texttt{Lbl}_\mathcal{V}$ and $\texttt{Lbl}_\mathcal{E}$ are the labeling relations for vertices

(i.e., relating one label to each vertex) and edges (i.e., one label for each edge) respectively, and $\mathcal{W}_\mathcal{E} \subset E \times [-w_{max}; w_{max}]$ is a weight assignment relation for edges.[5] Label l_v is the label of $v \in \mathcal{V}$ if $(v, l_v) \in \texttt{Lbl}_\mathcal{V}$, and represents a name of a schema element. Similarly $l_e \in \{\text{“}ref\text{''}, \text{“}value\text{''}, \text{“}type\text{''}, \text{“}pattern\text{''}\}$ is a label of edge $e \in \mathcal{E}$ if $(e, l_e) \in \texttt{Lbl}_\mathcal{E}$, and describes the function of the edge.

3.1.1 Basic Graph Construction:

Let $\mathcal{R}$ be a relational schema, $\mathcal{O}$ an ontology.

Basic nodes (vertices) and edges for *IncGraph+* are based on input schema elements, i.e., tables and attributes for *IncGraph+* ($\mathcal{R}$), or classes and properties for *IncGraph+* ($\mathcal{O}$).

Relational Schemata (*IncGraph+* ($\mathcal{R}$)): Let T the set of tables (relations) in the schema, A_t the set of attribute of table $t \in T$, $P \subset \{(t_1, a_1, t_2, a_2) | t_1, t_2 \in T, a_1 \in A_{t_1}, a_2 \in A_{t_2}\}$ the set of non-compound referential constraints between tables in $\mathcal{R}$. Then:

(Table Nodes) $t \in T \rightarrow v_t \in \mathcal{V} \wedge v_t.type = \text{”}T\text{”} \wedge (v_t, name(t)) \in \texttt{Lbl}_\mathcal{V}$

(Attribute Nodes) $a \in A_t \wedge t \in T \rightarrow v_a \in \mathcal{V} \wedge v_a.type = \text{”}A\text{”}$
$\wedge (v_a, name(a)) \in \texttt{Lbl}_\mathcal{V} \wedge e_a = (v_t, v_a) \in \mathcal{E}_d \wedge (e_a, \text{”}val\text{”}) \in \texttt{Lbl}_\mathcal{E}$

(Datatype Nodes) $v_a \in \mathcal{V} \wedge v_a.type = \text{”}A\text{”} \rightarrow v_{dt} \in \mathcal{V} \wedge v_{dt}.type = \text{”}D\text{”}$
$\wedge (v_{dt}, name(dt(v_a))) \in \texttt{Lbl}_\mathcal{V} \wedge e_{dt} = (v_a, v_{dt}) \in \mathcal{E}_d \wedge (e_{dt}, \text{”}type\text{”}) \in \texttt{Lbl}_\mathcal{E}$

(Reference Nodes) $(t_1, a_1, t_2, a_2) \in P \rightarrow v_p \in \mathcal{V} v_p.type = \text{”}P\text{”}$
$\wedge (v_p, name(a_1)) \in \texttt{Lbl}_\mathcal{V} \wedge e_{p_1} = (v_{t_1}, v_p) \in \mathcal{E}_u \wedge (e_{p_1}, \text{”}ref\text{”}) \in \texttt{Lbl}_\mathcal{E}$
$\wedge e_{p_2} = (v_{t_2}, v_p) \in \mathcal{E}_u \wedge (e_{p_2}, \text{”}ref\text{”}) \in \texttt{Lbl}_\mathcal{E}$

Ontologies (*IncGraph+* ($\mathcal{O}$)): Let C, D_P, O_P the set of class axioms, datatype property axioms and object property axioms in $\mathcal{O}$, respectively, and X be the set of OWL datatypes. Then:

(Class Nodes) $c \in C \rightarrow v_c \in \mathcal{V} \wedge v_c.type = \text{”}T\text{”} \wedge (v_c, name(c)) \in \texttt{Lbl}_\mathcal{V}$

(Datatype Property Nodes) $d \in D \wedge c = domain(d) \in D_P$
$\rightarrow v_d \in \mathcal{V}, v_d.type = \text{”}A\text{”} \wedge (v_d, name(d)) \in \texttt{Lbl}_\mathcal{V} \wedge e_d = (v_c, v_d) \in \mathcal{E}_d$
$\wedge (e_d, \text{”}val\text{”}) \in \texttt{Lbl}_\mathcal{E}$

(Datatype Range Nodes) $v_d \in \mathcal{V} \wedge v_d.type = \text{”}A\text{”} \wedge r = range(v_d) \in X$
$\rightarrow v_x \in \mathcal{V} \wedge v_x.type = \text{”}D\text{”} \wedge (v_x, name(r)) \in \texttt{Lbl}_\mathcal{V} \wedge e_x = (v_d, v_x) \in \mathcal{E}_d$
$\wedge (e_x, \text{”}type\text{”}) \in \texttt{Lbl}_\mathcal{E}$

[5] We denote optional edges with negative weights; they can be activated during matching by multiplying their weight with −1.

(Object Property Nodes) $p \in P \wedge d = domain(p) \wedge r = range(p)$
$\rightarrow v_p \in \mathcal{V} v_p.type = "P" \wedge (v_p, name(p)) \in \texttt{Lbl}_{\mathcal{V}} \wedge e_{p_1} = (v_d, v_p) \in \mathcal{E}_u$
$\wedge (e_{p_1}, "ref") \in \texttt{Lbl}_{\mathcal{E}} \wedge e_{p_2} = (v_r, v_p) \in \mathcal{E}_u \wedge (e_{p_2}, "ref") \in \texttt{Lbl}_{\mathcal{E}}$

3.2 Reasoning

We extend *IncGraph+* using reasoning on $\mathcal{O}$ in two ways. First, we assume that a reasoner infers and materializes relevant axioms prior to *IncGraph+* generation (basic reasoning). Second, we derive implicit information using custom, non-standard reasoning rules and directly encode consequences into *IncGraph+* ($\mathcal{O}$). This concerns aspects that may result in additional relevant matches with *IncGraph+* ($\mathcal{R}$) of a database schema: domains and ranges of properties other than the ones explicitly asserted, and inverse properties.

3.2.1 Sub Classes/Super Classes:

Domains and ranges are often modeled in a database at a granularity other than the one expressed in a corresponding ontology. In one case, this may be one or more specific sub class(es) of the actual domain or range, if the database is designed to accept only information about those specific sub classes. In another case, however, this can even be a *super* class of the domain or range, following the reasoning that *some* of the individuals of the super class can have values of that property. Although somewhat less frequent, databases occasionally happen to model properties in such an overly generic way.

In order to solve this, we add reference edges to the graph to encode all alternative connections. To express the fact that some cases are less expected, we assign weight factors to the additional edges:

(Sub Classes) $v \in \mathcal{V} \wedge v_d.type \in \{"A", "P"\} \wedge e = (v, v') \in \mathcal{E}_u \wedge v'.type = "C" :$
$\forall s \in C.subClass(class(v), s) \rightarrow e' = (v, s) \in \mathcal{E}_u \wedge (e', "ref") \in \texttt{Lbl}_{\mathcal{E}}$

(Super Classes) $v \in \mathcal{V} \wedge v_d.type \in \{"A", "P"\} \wedge e = (v, v') \in \mathcal{E}_u \wedge v'.type = "C" :$
$\forall s \in C : superClass(class(v), s) \rightarrow e' = (v, s) \in \mathcal{E}_u \wedge (e', "ref") \in \texttt{Lbl}_{\mathcal{E}} \wedge$
$(e', 0.5) \in \mathcal{W}_{\mathcal{E}}$

3.2.2 Pseudo Equivalence:

While real equivalences can be made explicit by a reasoner in pre-processing, another more subtle notion of equivalence may apply on *IncGraph+* on axioms that are not actually equivalent in the ontology: pseudo-equivalence. As relations and referential constraints have no semantic direction and because in *IncGraph+* we only model aspects that can find correspondences, properties also lose their direction in *IncGraph+*. This means, that inverse properties become effectively equivalent w.r.t. *IncGraph+*. However, they are not in the

underlying ontology and thus are modeled twice during graph construction. Consequently, they compete for matches and distract structural alignment.

We solve this issue by unifying pseudo-equivalent property axioms into a single node during *IncGraph+* construction, but maintain both labels for alternative lexical matching.

3.3 Patterns and Meta Knowledge

Due to different data modeling approaches, the same information is structured differently in relational databases and ontologies, and direct correspondences between modeling constructs are often difficult to establish: e.g., relational database schemata do not model concept and property hierarchies explicitly, many-to-many relations are expressed using intermediate tables, etc.: for example, typically three different ways are used in which an ontological *subClassOf* relation can be implemented in a relational model (se Fig. 4). In the database community, research has identified common design patterns for modeling interrelated data [GMUW08], and the Semantic Web research in turn noted various common ways for matching these database schema structures with the semantically equivalent ontology constructs [Se12, HM13].

Such common correspondence patterns provide valuable background knowledge helping to generate mappings between ontologies and relational databases: a set of initial mappings can be reinforced if they appear to satisfy a common pattern.

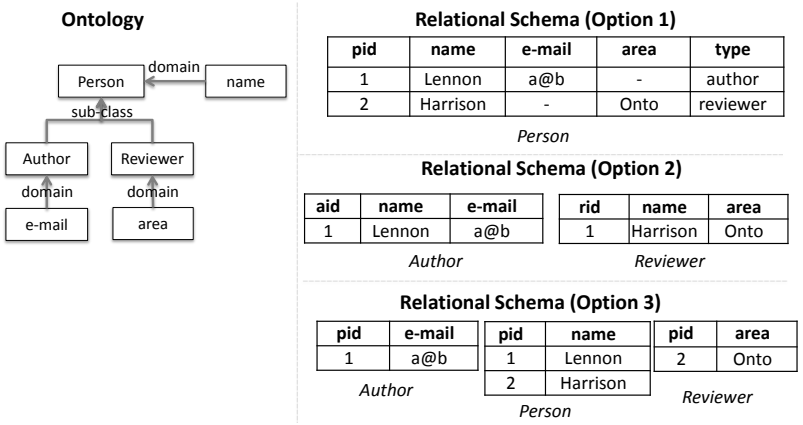


Fig. 4: Correspondence patterns: class-subclass hierarchy

Based on these common structures, we selected the following patterns, which can directly be used :

- *rdfs:subClassOf* relations between classes:
  - *One common table for all*. An example is shown as Option 1 in Fig. 4. Here one table stores information about all people: both authors and reviewers. The

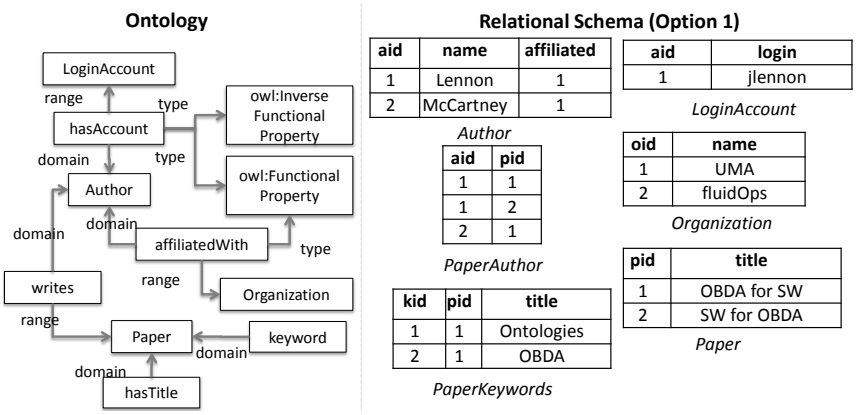


Fig. 5: Correspondence patterns: properties

type column stores the type of the record, and the type-specific fields, such as *e-mail* and *area* receive NULL values if they are not relevant for the record.

- *Separate unrelated tables*. Option 2 in Fig. 4 provides an illustration: authors and reviewers are stored in separate tables containing only fields relevant for the specific type. Common fields such as *name* are contained in both tables.
- *Separate tables linked via a 1:1 foreign key relation*. See Option 3 in Fig. 4: common fields are defined in the Person table which has 1:1 foreign key relations to both category-specific ones.

- *owl:ObjectProperty* links:
  - *1:1 relation*: two tables are connected via their unique keys, similarly to the relation between the tables Author and LoginAccount in Fig. 5.
  - *1:n relation*: two tables are connected via a foreign key, which is unique in one of the tables, as is the Organization table in Fig. 5. The unique key *oid* of the Organization table is referenced in the Author table.
  - *n:m relation*: two tables are connected via an intermediate table containing two foreign key columns. The PaperAuthor table in Fig. 5 is used as such an intermediary between the Author and Paper tables.

- *owl:DatatypeProperty* links:
  - *1:1 relation*: a column in the table directly contains the value of a datatype functional property (e.g., *hasTitle*).
  - *1:n relation*: a separate table is linked via a foreign key to the main table and contains an additional column for the data values. The example is the PaperKeywords table in Fig. 5.

IncMap exploits these patterns in heuristic rules that enrich the schema graphs. When building a schema graph, these rules add to the graphs special *pattern nodes*. Such pattern node represents a specific pattern type (e.g., "class-subClass") and is connected by edges to the relevant content nodes (tables and fields for the database schema graph, classes and properties for the ontology graph). Apparently, while patterns in the database are merely structural and may or may not represent the assumed semantics, their correspondences on the ontology side are factual axioms. In addition, some patterns are also ambiguous (e.g., 1:1 relations vs. the third subClass pattern). Also, a number of patterns cannot even be identified with certainty, but heuristics apply that result in varying confidence scores. Thus, on the database side, we employ weighted edges to connect pattern nodes, representing their detected confidence score. The role of the pattern nodes is to reinforce the connection between involved nodes on both sides and thus make the fixpoint computation algorithm more likely to lead to higher similarity scores for pairs of nodes in the neighborhood of corresponding or aligned pattern node.

Formally, for each supported pattern on relational schema $\mathcal{R}$, there is a pattern qualifier heuristic H that assigns each subset of schema elements (i.e., each $E \in \mathcal{P}(\mathcal{R})$) a score, denoting the confidence they might form the specified pattern.

Then: $H(E) > 0.0 \rightarrow \forall el \in E : v_x \in \mathcal{V} \wedge v_x.type = "X" \wedge e = (v_{el}, v_x) \in \mathcal{E}_d \wedge e = (v_{el}, v_x) \in \texttt{Lbl}_{\mathcal{V}}$

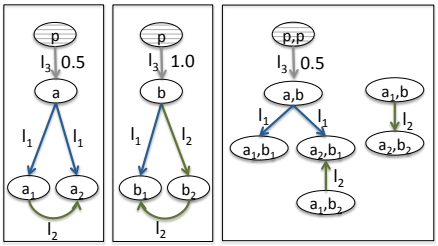
4 Matching & Mapping

IncMap produces mappings in two main stages: (1) matching, i.e., finding correspondences between nodes in schema graphs of the relational source and the target ontology, and (2) mapping generation based on matched correspondences.

4.1 Matching Process

Matching a source and target *IncGraph+* starts with calculating the cartesian products between nodes in the source and target, for each respective node type. Each pair, i.e., each potential correspondence, is then evaluated by an initial match operator, which calculates an initial lexical similarity between the nodes based on their labels. The preferred initial match operator in *IncMap* is word-bag Jaccard similarity on stemmed, stop-word filtered tokenizations of the labels. For nodes with no lexical labels, such as pattern nodes, this operator assigns an initial similarity of 0.5.

Next, paired nodes get reassembled into a new graph, based on common edges that both paired nodes shared in their respective *IncGraph+*. This is in preparation of a following phase, where a fixpoint computation based on the Similarity Flooding algorithm [MGMR02] refines the initial scores based on structural similarities. In [MGMR02], the reassembled graph used as input for Similarity Flooding is called a Pairwise Connectivity Graph, (PCG). Our graph is based on PCG but extended in several ways, primarily to accommodate



(a) Source (b) Target (c) PCG+

Fig. 6: Construction of Extended Pairwise Connectivity Graph (Simplified)

undirected edges, weighted edges and optional edges, which are not supported in the original PCG. We thus refer to this reassembled graph as Extended Pairwise Connectivity Graph (PCG+). Figure 6 shows a generic and simplified example of two input graphs and the resulting PCG+ (Figure 6c). The PCG+ has a node for every pair in the cross products of nodes from the input graphs, where nodes of some type (or color) in one graph will be paired only with nodes of the same color in the other. Therefore, in the figure, all combinations of a, a_1, a_2, b, b_1 and b_2 are in the graph, but only one node pairing p_1 with p_2. Edges are added to the PCG+ wherever both constituent nodes of a pair had a shared type of edge in the same direction. For instance, a and b have both an outgoing edge labeled l_1, which lead to a_1 and b_1, respectively. Therefore, the pair (a, b) in the PCG+ also has an outgoing edge l_1 leading to the pair (a_1, b_1). In addition, if either edge was weighted, their weights get multiplied for the PCG+. Unweighted edges are assumed to carry a weight of 1.0. If either edge is optional, the common edge becomes optional.

The PCG+ is then transformed into the final input for Similarity Flooding by adding inverse edges to avoid black holes in the fixpoint computation and by (re-)assigning weights to all edges to balance the score distribution.

Finally, the fixpoint computation will repeatedly distribute scores of matches to neighboring nodes depending on edge weights, thus refining the score of correspondences structurally. After each iteration, we check for optional edges supported by a pattern node with a top score, in which case the edge would be activated for the next iteration. The fixpoint computation halts if maximum score changes drop below a configurable maximal delta during one iteration or if a maximum number of iterations has been reached.

4.2 Mapping Generation

IncMap exports mappings based on most likely correspondences calculated during the fixpoint computation.

Correspondences are selected based on final scores. *IncMap* supports the interpretation of several correspondences as 1:1 mappings or n:1 mappings, but not n:1 or n:m mappings. Thus, intuitively, for each target side node (i.e., each node in the ontology *IncGraph+*), one

correspondence should be selected. We refer to this set of correspondences as the *target top-1* set of correspondences.

However, target top-1 correspondences may lead to significant inconsistencies, lowering the quality of the resulting mappings. For instance, the best match for a property will in some cases match its range class to a different table key then the one chosen as best match for the class node of the range. While this might even be correct on occasion – either, because the range class match is the one, which is incorrect, or because both matched keys define identical individuals – our general experience is that accepting those inconsistencies lowers mapping quality on average. Therefore, we do not select target top-1 correspondences but first choose target top-1 correspondences for classes only and then choose for properties from a restricted set that interprets domains and ranges consistently with the chosen class matches.

Finally, for each correspondence, one mapping rule is being generated using R2RML (RDB to RDF Mapping Language)[6] as mapping language. Technically, this is enabled by provenance information encoded within every node.

5 Experimental Evaluation

While all the extensions of *IncMap* discussed in this paper were motivated by putting our system into practice over the last few years in different application domains, we did run experiments to systematically analyze the impact of each extensions, also in comparison to other systems.

5.1 Benchmark Suite

We ran experiments using our mapping generation benchmark suite called RODI [Pi15]. RODI was designed to test *RDB2RDF* mappings in end-to-end mapping scenarios: it provides an input database and a target ontology and requests complete mappings that enable to execute queries over the target ontology while retrieving the results from the underlying source data. The effectiveness of mappings is then judged by a score that mainly represents the number of query tests that return the expected results on mapped data sources.

While RODI is extensible and can run scenarios in different application domains, it ships with a set of *default scenarios* that are designed to test a wide range of fundamental *RDB2RDF* mapping challenges in a controlled fashion.

5.2 Benchmark Results

In the following, we discuss the results of evaluating our system *IncMap*. We test two versions of *IncMap*, a basic setup with only basic reasoning (no custom reasoning rules)

[6] `http://www.w3.org/TR/r2rml/`

and without patterns (*IncMap basic*), and a complete version with all features combined (*IncMap*). We directly compare *IncMap* to its predecessor (called IncMap 1.0 [Pi13]), as well as to the best-performing systems from the original RODI benchmark experiments [Pi15], BootOX [Ji15], and -ontop- [Ro15]. We have also added experiments in a setup with COMA++ [Au05] as a well-known baseline for generic schema matching.

RODI Scenarios: As a first series of experiments we followed the RODI default scenarios. Most default scenarios are set in the conference domain, with two additional scenarios; one in the domain of geographic data as well as another in the oil & gas domain.

The default benchmark scenarios therefore provide nine different relational databases that are to be mapped to their respective corresponding target ontologies, which are provided as T-Boxes (i.e., only schema information with no data). Each benchmark scenario (i.e., each pair of a source database and a target ontology) is based on an existing pair of an ontology/database schema (CMT, CONFERENCE, and SIGKDD) as well as variants where the ontology/database schema was mutated to test a different set of integration challenges.

For example, "adjusted naming" scenarios in the conference domain are closely modeled after the original ontology, only with adjusted identifier names while the database schema was normalized to fourth normal form ($4NF$). "Restructured" scenarios are remodeled from ground up to follow widely used relational database design patterns, rather than following closely the patterns used in ontologies. Most significantly, this includes class hierarchies, resulting in cases where abstract parent classes have no corresponding table in the database, several sibling classes being jointly represented in a single table, etc. In addition, a selection of scenarios with advanced modeling patterns also forms part of the default scenarios: "missing FKs" represents the case of a database with no explicit referential constraints at all, and "denormalized" scenarios contain some denormalized tables. Finally, in the "Combined Cases" different mutations are mixed together (e.g., "adjusted naming" and "missing FKs"). Moreover, all non-conference scenarios use complex real-world relational schemata and ontologies of significant size.

Each scenario has different test categories that allow to break down results to, e.g., class mappings vs. property mappings, mappings based on simple 1:1 correspondences vs. more complex compositions, etc. RODI calculates scores between 0.0 and 1.0, based on the averages of per-test F-measures for each scenario and test category.

Results: Table 1 shows RODI scores for each default scenario on every tested system. RODI scores basically indicate the percentage of successfully passed query tests, although technically defined on the basis of averages of per-query F-measures.

For most scenarios, *IncMap* outperforms all other systems with varying margins. In particular, *IncMap* outperforms its predecessor (IncMap 1.0) by far, improving in all case and even rising by a factor of 10 and more in some of the more complex cases (SIGKDD combined, CONFERENCE missing FKs). Numbers also indicate that *IncMap* has successfully overcome architectural issues in graph construction that had caused its

Tab. 1: Average results of all tests per scenarios (scores based on F-measure). Best numbers per scenario in bold print.

Scenario	IncMap 1.0	-ontop-	BootOX	COMA++	*IncMap* basic	*IncMap*
	Conference adjusted naming					
CMT	0.45	0.28	**0.76**	0.48	0.45	0.66
CONFERENCE	0.26	0.26	0.51	0.36	0.56	**0.64**
SIGKDD	0.21	0.38	0.86	0.66	0.79	**0.90**
	Conference restructured					
CMT	0.38	0.14	0.41	0.38	0.45	**0.64**
CONFERENCE	0.16	0.13	0.41	0.31	0.46	**0.56**
SIGKDD	0.11	0.21	0.52	0.41	0.45	**0.69**
	Conference combined case					
SIGKDD	0.05	0.21	0.48	0.28	0.45	**0.55**
	Conference missing FKs					
CONFERENCE	0.03	-	0.33	0.21	**0.41**	**0.41**
	Conference denormalized					
CMT	0.22	0.20	0.44	-	0.52	**0.71**
	Geodata					
Geodata	0.00	-	**0.13**	-	0.08	0.08
	Oil & Gas					
Oil & Gas	0.06	0.10	0.14	0.02	0.12	**0.17**

predecessor to fall way behind the competition in different cases. This mostly affects scenarios involving ontologies that are rather expressive (CONFERENCE and SIGKDD, as opposed to CMT).

Between the two versions of *IncMap*, the *basic* version and complete *IncMap*, a positive impact of the advanced patterns and reasoning in the full version can generally be observed. The only exception is "CONFERENCE Missing FKs", which is natural as both reasoning and patterns rely on referential constraints to produce additional correspondences with the database.

In general, it can however be observed that *IncMap*, even while outperforming other systems, still significantly struggles with the more complex scenarios as well as the two real-world cases (Geodata and Oil & Gas). The further the database schema deviates from its corresponding ontology by using traditional relational database design patterns (restructured, advanced scenarios), the more mapping result quality drops.

6 User Study

Next to systematic experiments with our benchmark suite RODI, we have conducted a user study to evaluate the merits of *IncMap* in practice. We have therefore asked a number of *RDB2RDF* data integration professionals to perform a small, but real-world, industry mapping problem with and without automatically bootstrapped mappings, did ask for their impressions and observed how long the task took them.

6.1 Study Design

We have set the study problem in a commercial application domain that we frequently see in our everyday consulting practice with customers at fluid Operations: data center management. The domain comprises information about hardware, software and services in large data centers, monitoring information etc. As relational data source for the study, we have used one realistic customer database of advanced complexity, containing 61 tables with a combined 776 columns. The target ontology was a subset of the enterprise datacenter/cloud vocabulary that we normally use for commercial data integration projects in this domain. The full vocabulary is publicly available[7]. We did limit the mapping task to mapping information regarding *hosts*. This limits the actually relevant part of the target ontology to 3 classes, 8 datatype properties and 4 object properties. While scaled down, this setup closely resembles our typical real-world data integration projects insofar as it requires to map parts of a holistic ontology from only a small subset of information in a large source database. To further specify the task we have provided a set of dashboard queries for which the participants needed to curate the mapping.

We have asked 7 randomly selected data integration experts to create the required mapping, either starting with an intial, automatically bootstrapped mapping using *IncMap*, or from scratch. We then measured the time taken to complete the mapping to each expert's satisfaction in both cases. We asked all users to first perform the task using *IncMap* mappings. Note, that this puts *IncMap* at a disadvantage over the from-scratch case, as any time taken to get basically familiarized with the source schema would count against the first task.

Additionally, we did ask users before starting their first task to judge the automatically generated mappings by their perceived utility without any corrections. We also asked after the experiments about the preferred approach for future tasks. Specifically, we did ask the following two questions:

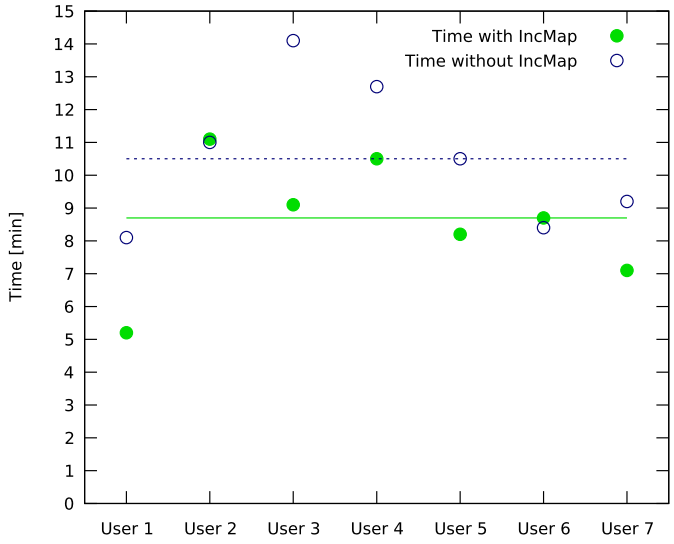
Question 1 (before first task): "When inspecting the initially bootstrapped IncMap mappings, how useful do you think they are on a scale from 'not useful at all' to 'highly useful'?"

Question 2 (after last task): "When performing a similar task again in the future, would you prefer to work with bootstrapped IncMap mappings or start from scratch?"

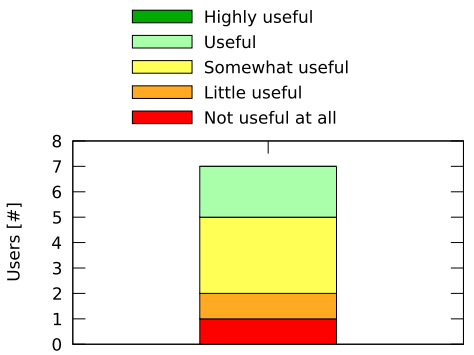
6.2 Results and Observations

Figure 7 shows our observations from the user study. First, Figure 7a depicts the times taken to complete the mapping with and without *IncMap* bootstrapping for each of the users. The green line marks the median time taken with *IncMap* bootstrapping, while the dashed blue

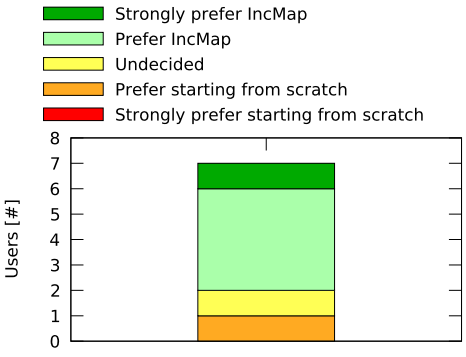
[7] `http://www.fluidops.com/en/company/training/open_source`



(a) Times taken to complete



(b) Question 1 (before task)



(c) Question 2 (after task)

Fig. 7: User Study Results

line marks the median time taken without *IncMap*. It shows that, although all users did first perform the task on *IncMap* bootstrapped mappings and thus were already familiar with the source schema when trying again from scratch, it took all but two of them less time to complete the mapping when working with *IncMap*, with the two exceptions reporting very small differences. On average, despite this disadvantage for *IncMap* in the experiment, users did save 121 seconds (roughly 20%), with a difference of 108 seconds between median times.

Figure 7b shows the impressions of users regarding the *initial* automatic mappings: although some users did acknowledge them to be somewhat useful, results are poor on average. This is in line with expectations, as fully automatic mappings on relatively large schemata are known to be generally difficult to use without prior corrections, due to both incomplete matches and a high number of false positives (e.g., [HQ07]).

Hence, the more relevant question is whether the mappings are useful as a starting point for manual editing. Besides the obvious indication in this direction from task completion times, we also did ask users about their *subjective* judgment, namely which method of building methods they would prefer next time for a similar task. The responses are depicted in Figure 7c: a clear majority of 5 out of 7 users would prefer automatically bootstrapped mappings from *IncMap*, with another one neutral and only one user leaning slightly against.

7 Related Work

IncMap automatically generates mappings between relational databases and ontologies in R2RML, a recent W3C recommendation. Our approach builds on top of previous research on database patterns [Se12, HM13], lexical (e.g., [CH13]) and structural [MGMR02] ontology matching techniques, and introduces original graph construction algorithms.

A number of other mapping generation systems have been developed or updated to support R2RML. QODI [TSM13] generates mappings with a rather simple mechanism but additionally exploits the query workload to improve mappings. The approach of [Ji15] relies on the ontology alignment tool LogMap [JRG11] after transforming the database schema into an ontology representation in a naive pre-processing step. Similar setups have been built in conjunction with -ontop- [Ro15] and MIRROR [de15]. MIRROR generates specialized mappings that consider relational-to-ontology mapping patterns, but produces a target-agnostic ontology and thus requires ontology alignment to map to a target ontology. Karma [Kn12] employs an interactive, semi-automatic approach to produce R2RML mappings directly from input relational schemata but produces no fully automatic (i.e., initial) mappings. [Ne13] also produces R2RML but requires an extreme level of human interaction and generates only final mappings automatically, after correspondences have been specified.

Other related systems predate current *RDB2RDF* approaches as well as the R2RML language. Most prominently, COMA++ [Au05] was originally designed to match relational schemata but has evolved to also perform inter-model matchings, although it is not its main focus. Other previous efforts towards *RDB2RDF* mapping generation (e.g., [Pa06]) date back to

the early days of OBDI, and work has since been discontinued. A comprehensive overview of *RDB2RDF* efforts, including related approaches of automatic mapping generation, can be found in the survey of Spanos et al. [SSM12].

To evaluate and compare *IncMap*, we use RODI [Pi15], a recently released *RDB2RDF* benchmark suite. RODI is designed to measure the quality of mapping generation end-to-end by testing queries on the mapping result, and specifically targets mapping challenges that arise from the inter-model gap. No other benchmark for this setting exists to the best of our knowledge. A number of papers discuss related quality aspects from a more general and theoretic perspective (e.g., [CL14, MC14]). However, theoretical criteria such as consistency and completeness give no indication of whether mapping results reflect the expected semantics and lead to useful query results.

8 Conclusion

We have presented *IncMap*, a system to automatically generate direct mappings between relational databases and given target ontologies. *IncMap* is based on an intermediate internal graph representation that allows the representation of both factual knowledge and heuristically observed patterns from the input. The graph model supports a combination of lexical and structural matching for initial alignment.

To evaluate *IncMap*, we experimentally compared it to other systems using RODI [Pi15], a recent benchmark for automatically generated *RDB2RDF* mappings. Results demonstrate that *IncMap* does not only improve massively over its predecessor *IncMap*, but also outperforms all other tested systems. We have also conducted a user study that demonstrates *IncMap* to be useful in practice.

Future work will include support for n-way joins in mapping object properties (for $n > 3$) and improved pattern heuristics.

References

[Au05] Aumueller, David; Do, Hong-Hai; Massmann, Sabine; Rahm, Erhard: Schema and Ontology Matching with COMA++. In: SIGMOD. 2005.

[Bh15] Bhardwaj, Anant P.; Bhattacherjee, Souvik et al.: DataHub: Collaborative Data Science & Dataset Version Management at Scale. CIDR, 2015.

[CH13] Cheatham, Michelle; Hitzler, Pascal: String Similarity Metrics for Ontology Alignment. In: ISWC. 2013.

[CL14] Console, Marco; Lenzerini, Maurizio: Data Quality in Ontology-Based Data Access: The Case of Consistency. In: AAAI. 2014.

[de15] de Medeiros, LF. et al.: MIRROR: Automatic R2RML Mapping Generation from Relational Databases. In: ICWE. 2015.

[DS13] Dong, Xin Luna; Srivastava, Divesh: Big Data Integration. PVLDB, 6(11):1188–1189, 2013.

[GMUW08] Garcia-Molina, Hector; Ullman, Jeffrey D.; Widom, Jennifer: Database Systems – The Complete Book. Prentice Hall, 2008.

[HM13] Hornung, Thomas; May, Wolfgang: Experiences from a TBox Reasoning Application: Deriving a Relational Model by OWL Schema Analysis. In: OWLED Workshop. 2013.

[HQ07] Hu, Wei; Qu, Yuzhong: Discovering Simple Mappings Between Relational Database Schemas and Ontologies. In: ISWC/ASWC. 2007.

[Ji15] Jiménez-Ruiz, E. et al.: BootOX: Practical Mapping of RDBs to OWL 2. In: ISWC. 2015.

[JRG11] Jiménez-Ruiz, Ernesto; Grau, Bernardo Cuenca: LogMap: Logic-Based and Scalable Ontology Matching. In: International Semantic Web Conference. 2011.

[Kn12] Knoblock, C. et al.: Semi-Automatically Mapping Structured Sources into the Semantic Web. In: ESWC. 2012.

[MC14] Mora, Jose; Corcho, Oscar: Towards a Systematic Benchmarking of Ontology-Based Query Rewriting Systems. In: ISWC. 2014.

[MGMR02] Melnik, Sergey; Garcia-Molina, Hector; Rahm, Erhard: Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In: ICDE. 2002.

[Ne13] Neto, L.E.T. et al.: R2RML by Assertion: A Semi-automatic Tool for Generating Customised R2RML Mappings. In: ESWC (Satellite Events). 2013.

[Pa06] Papapanagiotou, P. et al.: Ronto: Relational to Ontology Schema Matching. AIS SIGSEMIS BULLETIN, 2006.

[Pi13] Pinkel, Christoph; Binnig, Carsten; Kharlamov, Evgeny; Haase, Peter: IncMap: Pay-as-you-go Matching of Relational Schemata to OWL Ontologies. In: OM. 2013.

[Pi15] Pinkel, Christoph; Binnig, Carsten; Jimenez-Ruiz, Ernesto et al.: RODI: A Benchmark for Automatic Mapping Generation in Relational-to-Ontology Data Integration. In: ESWC. 2015.

[Ro15] Rodriguez-Muro, M. et al.: Efficient SPARQL-to-SQL with R2RML mappings. Journal of Web Semantics, 2015.

[Se12] Sequeda, J. et al.: Relational Database to RDF Mapping Patterns. In: WOP. 2012.

[SSM12] Spanos, Dimitrios-Emmanuel; Stavrou, Periklis; Mitrou, Nikolas: Bringing Relational Databases into the Semantic Web: A Survey. Semantic Web, 3(2), 2012.

[TSM13] Tian, Aibo; Sequeda, Juan F; Miranker, Daniel P: QODI: Query as Context in Automatic Data Integration. In: ISWC. 2013.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 165

Anfrage-getriebener Wissenstransfer zur Unterstützung von Datenanalysten

Andreas M. Wahl[1] Gregor Endler[1] Peter K. Schwab[1] Sebastian Herbst[1] Richard Lenz[1]

Abstract: In größeren Organisationen arbeiten verschiedene Gruppen von Datenanalysten mit unterschiedlichen Datenquellen, um analytische Fragestellungen zu beantworten. Das Formulieren effektiver analytischer Anfragen setzt voraus, dass die Datenanalysten profundes Wissen über die Existenz, Semantik und Verwendungskontexte relevanter Datenquellen besitzen. Derartiges Wissen wird informell innerhalb einzelner Gruppen von Datenanalysten geteilt, jedoch meist nicht in formalisierter Form für andere verfügbar gemacht. Mögliche Synergien bleiben somit ungenutzt. Wir stellen einen neuartigen Ansatz vor, der existierende Datenmanagementsysteme mit zusätzlichen Fähigkeiten für diesen Wissenstransfer erweitert. Unser Ansatz fördert die Kollaboration zwischen Datenanalysten, ohne dabei etablierte Analyseprozesse zu stören. Im Gegensatz zu bisherigen Forschungsansätzen werden die Analysten beim Transfer des in analytischen Anfragen enthaltenen Wissens unterstützt. Relevantes Wissen wird aus dem Anfrageprotokoll extrahiert, um das Auffinden von Datenquellen und die inkrementelle Datenintegration zu erleichtern. Extrahiertes Wissen wird formalisiert und zum Anfragezeitpunkt bereitgestellt.

Keywords: Datenintegration, Kollaboration, Anfrageverarbeitung

1 Einführung

Zur Beantwortung von analytischen Fragestellungen sind meist mehrere heterogene Datenquellen erforderlich. Die benötigten Daten sind oftmals semi-strukturiert und liegen heutzutage nicht mehr ausschließlich in relationaler Form vor, sondern werden in den verschiedensten Formaten und Systemen gespeichert. So fallen beispielsweise bei der Patientenüberwachung in der Intensivmedizin neben strukturierten Patientenstammdaten auch große Mengen semi-strukturierter Medikationspläne, Datenströme von Vitalparametern oder Freitextnotizen an [Sa11]. Um solche Daten, unter anderem für die klinische Forschung, effizient auswertbar zu machen, können neuartige Ansätze wie die von Stonebraker et al. vorgeschlagenen Polystores verwendet werden [St15]. Ein Polystore ermöglicht es, Daten aus heterogenen Datenquellen in einer einzigen Anfrage miteinander zu verknüpfen.

Auch wenn dadurch die prinzipielle Auswertbarkeit dieser Daten gegeben ist, müssen die verschiedene Datenquellen immer noch weitgehend manuell von Datenanalysten zusammengeführt werden. Die Fragestellungen, die von den Datenanalysten beantwortet werden sollen, beziehen sich naturgemäß auf deren mentales Modell der Problemdomäne. Dieses sich ständig weiterentwickelnde Modell beinhaltet Informationen darüber, wann Datenquellen nützlich sind, wie diese verknüpft werden können, wie deren Inhalt zu interpretieren ist oder welches Vokabular verwendet wird. Derartige Wissensaspekte werden vielfach nicht

[1] FAU Erlangen-Nürnberg, Lehrstuhl für Informatik 6 (Datenmanagement), ErsterVorname.Nachname@fau.de

dokumentiert, sondern unter Kollegen informell weitergegeben. Besonders in größeren Organisationen arbeiten häufig mehrere Teams von Datenanalysten an der Beantwortung unterschiedlicher Fragestellungen. Wenn diese Teams nicht miteinander interagieren, verpassen sie wichtige Gelegenheiten, ihr Wissen über den Datenbestand zu teilen oder von den Erfahrungen anderer zu profitieren.

Aus diesem Grund entwickeln wir ein *Anfrage-getriebenes Wissenstransfersystem (AWTS)*. Ein AWTS erweitert ein Datenmanagementsystem, beispielsweise einen Polystore, um neuartige Dienste, die Wissen aus Anfragen formalisieren und allen Benutzern zur Verfügung stellen können. Dabei wird ausgenutzt, dass das einer Anfrage zugrundeliegende mentale Modell partiell extrahiert und analysierbar gemacht werden kann. Das System unterstützt die Abbildung dieses Modells auf den tatsächlichen Datenbestand, indem die bisherigen Abbildungen anderer, ähnlicher Anfragen wiederverwendet werden.

2 Anfrage-getriebener Wissenstransfer

Die Schaffung von neuem Wissen spielt für den Alltag von Datenanalysten eine große Rolle. Im Folgenden bezeichnen wir mit dem Begriff *Wissen* Domänenwissen über verfügbare Datenquellen, das benötigt wird, um die Quellen für Analysezwecke zu verwenden. Dieses Wissen umfasst unter anderem folgende Aspekte: **(1)** Welche Datenquellen sind verfügbar? **(2)** Welche Teile dieser Datenquellen sind für welche Zwecke geeignet? **(3)** Welches Vokabular und welche Semantik wird zur Beschreibung des Inhalts bestimmter Datenquellen verwendet? **(4)** Wie können Datenquellen miteinander verknüpft werden? **(5)** Welche relevanten Informationen können aus dem Inhalt der Datenquellen abgeleitet werden? **(6)** Wer verwendet welche Datenquellen in welchem zeitlichen Kontext?

Wir beziehen uns auf das SEKI-Modell [NK98], um zu erläutern, wie ein AWTS die Schaffung von neuem Wissen unterstützen kann (Abb. 1). Das Modell unterscheidet zwischen *explizitem* und *stillem* Wissen. Ersteres kann direkt formuliert und geteilt werden, wohingegen letzteres persönliche Fähigkeiten und mentale Modelle umfasst, die nicht direkt formalisiert werden. Stilles Wissen wird durch *Sozialisierung* weitergegeben. Dieses Wissen wird durch *Externalisierung* explizit gemacht, beispielsweise durch textuelle Dokumentation. Durch *Kombination* wird neues Wissen durch Analyse des verfügbaren expliziten Wissens geschaffen. Bei der *Internalisierung* wird explizites Wissen von Individuen verinnerlicht und in stilles Wissen umgewandelt.

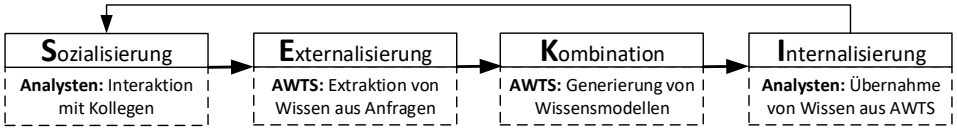


Abb. 1: SEKI-Modell am Beispiel von anfrage-getriebenem Wissenstransfer

Datenanalysten wenden ihre Erfahrungen und mentalen Modelle an, um analytische Fragestellungen zu formulieren. Stilles Wissen wird bereits durch Interaktion innerhalb von Analystenteams geteilt. Ein AWTS fördert die Externalisierung dieses Wissens, indem alle analytischen Anfragen innerhalb einer Organisation gesammelt werden. Teile der mentalen Modelle der Analysten werden somit in strukturierter Form erfasst und können explizit

gemacht werden. Das AWTS übernimmt die Kombination dieses Wissens durch die Generierung von Wissensmodellen, die Muster, Beziehungen und Hinweise zur Semantik von Datenquellen aus den Anfragen ableiten. Auf Basis dieser Modelle erhalten die Analysten Empfehlungen über Datenquellen. Zur Internalisierung dieses neuen Wissens können die Empfehlungen analysiert und in zukünftigen Anfragen berücksichtigt werden.

2.1 Vorteile der Verwendung eines AWTS aus Sicht der Datenanalysten

Um die Vorteile eines AWTS für Datenanalysten zu verdeutlichen, erläutern wir dessen Verwendung anhand eines praxisnahen Szenarios aus der klinischen Forschung. Wir betrachten drei Teams von Datenanalysten, die ein zentralisiertes AWTS verwenden (Abb. 2). Das AWTS verwaltet verschiedene Daten aus elektronischen Krankenakten, die zusätzlich für die klinische Forschung genutzt werden sollen („secondary use“[2]). Zum besseren Verständnis werden alle Beispielanfragen stark vereinfacht dargestellt.

Team 1 verwendet Medikationspläne aus der Datenquelle D1 (Speicherformat JSON) und anonymisierte Patientendaten aus D2 (relationale Speicherung), um die Dosierung von Medikamenten zu analysieren. Team 2 führt ebenfalls Analysen der Medikationsgabe durch, verlässt sich dabei aber neben D2 auf die Medikationspläne aus D3 (Speicherformat CSV) sowie D4 (relationale Speicherung). Team 3 ist auf die Zeitreihenanalyse der Messwerte von Patientenmonitoren spezialisiert. Es verwendet dazu die Datenquellen D5 (Speicherformat Avro) und D6 (Speicherformat Parquet) aus einem verteilten Dateisystem. Initial wissen die Teams nicht voneinander. Das AWTS bietet eine einheitliche Anfrageschnittstelle für verschiedene Arten von Datenquellen (im Beispiel SQL). Anfragen werden beim Zugriff auf die Datenquellen entsprechend übersetzt.

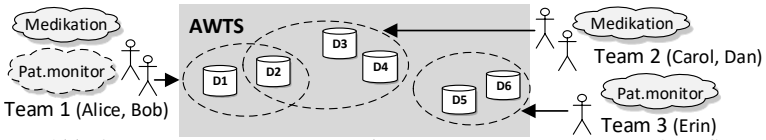


Abb. 2: Drei Teams von Datenanalysten interagieren mit einem AWTS

Während Alice von Team 1 auf die Ergebnisse einer Anfrage (Abb. 3) wartet, erkennt das AWTS auf Basis bestimmter Anfragen von Team 2 (Abb. 4), dass beide Teams die Datenquelle D2 verwenden. Über eine graphische Schnittstelle stellt das AWTS für Alice nun Informationen und Empfehlungen für zukünftige Anfragen auf Basis des Wissens von Team 2 bereit. Durch Analyse der Anfragen von Team 2 erkennt das AWTS, dass D3 und D4 bereits mit D2 verknüpft worden sind. Daher präsentiert es Alice direkt eine einheitliche Sicht auf diese Datenquellen. Um Alice bei der Analyse von D3 und D4 zu unterstützen, gibt das AWTS unter anderem Auskunft über die Verwendungshäufigkeit und den Verwendungskontext in den bisherigen Analysesitzungen von Team 2.

```
SELECT D2.id, D2.Abteilung
FROM D1 JOIN D2 ON D2.id = D1.PatNr
WHERE D1.Wirkstoff LIKE 'Dexametha%';
```

Abb. 3: Anfrage von Alice (Team 1)

```
SELECT MIN(D2.Alter)
FROM D4 JOIN D2 USING id
WHERE D4.Arzneistoff = 'Salbutamol';
```

```
SELECT D2.id, D3.Substance, D3.Dose
FROM D3 JOIN D2 ON D2.id = D3.Patient;
```

Abb. 4: Anfrageprotokollauszug von Team 2

[2] Vgl. zu diesem Thema auch Arbeiten der zugeh. AG der GMDS: http://www.pg-ss.imi.uni-erlangen.de/

Bob von Team 1 hat kürzlich die Aufgabe erhalten herauszufinden, inwiefern die Gabe einiger Wirkstoffe mit dem Auftreten kritischer Werte bestimmter Vitalparameter korreliert. Er kennt noch keine Datenquellen, die diese Parameter enthalten, aber er hat eine ungefähre Vorstellung von den Daten, die er sucht. Er verwendet sein mentales Modell, um eine Anfrage zu schreiben, welche die hypothetische Datenquelle `Vitalparameter` referenziert (Abb. 5). Bob geht davon aus, dass die gesuchte Datenquelle die Attribute `Herzfrequenz` und `Blutdruck` besitzt. Das AWTS verwendet seine Annahmen, um geeignete Datenquellen zu empfehlen. Durch Auswertung der Struktur der im Anfrageprotokoll referenzierten Datenquellen erkennt das AWTS, dass Erin von Team 3 schon einmal Datenquellen verwendet hat, die möglicherweise für Bob geeignet sind (Abb. 6). Daher schlägt das AWTS die Verwendung von D5 (Attribute `HeartRate, BloodPressure, Time, Patient`) und D6 (Attribute `HRt, BTemp, BP, TStmp, PId`) vor. Bob kann nun entscheiden, ob er den Empfehlungen folgen will. Wenn Bob dies tut, wird seine Anfrage vom AWTS derart modifiziert, dass an Stelle von `Vitalparameter` mindestens eine der gefundenen Datenquellen referenziert wird. Die von Erin verwendeten Umbenennungen werden zudem Teil einer Ontologie, mit deren Hilfe Bob das Vokabular anderer Teams einfacher erfassen kann. Über einen interaktiven Dialog kann er eine Rückmeldung bezüglich der Vorschläge an das AWTS übermitteln. Das AWTS speichert die Abbildung zwischen seinem mentalen Modell und den tatsächlich verfügbaren Datenquellen und erzeugt automatisch eine individuelle Sicht, die mit seinem mentalen Modell korrespondiert. Bob kann diese Sicht direkt in zukünftigen Anfragen verwenden, ohne dass zusätzliche Interaktionen mit dem AWTS notwendig sind.

```
SELECT Herzfrequenz, Blutdruck
FROM Vitalparemeter
WHERE Herzfrequenz >= 130;
```

Abb. 5: Anfrage von Bob (Team 1)

```
SELECT Patient, BloodPressure AS Blutdruck
FROM D5
WHERE HeartRate < 90 AND Time > '16-01-10';
-----------------------------------------------
SELECT HRt AS Herzfrequenz, BP AS Blutdruck
FROM D6
WHERE PId = 'P41' AND TStmp = '1475693932';
```

Abb. 6: Anfrageprotokollauszug von Erin (Team 3)

Sobald Synergien zwischen verschiedenen Teams identifiziert werden, können die Datenanalysten das Wissen anderer direkt in die formalisierte Repräsentation ihrer mentalen Modelle übernehmen. Das AWTS unterstützt somit eine vereinfachte Datenintegration. Zum Beispiel kann Bob das Wissen von Erin nutzen, solange er an Analysen von Vitalparametern beteiligt ist. Das bedeutet, dass er dieselbe Sicht auf die Vitalparameter erhält wie Erin selbst. Wenn er sich nicht mehr für diese Daten interessiert, kann er auch wieder zu seiner auf Medikationspläne fokussierten Sicht des Datenbestands zurückkehren.

2.2 Formalisierung von Wissen aus analytischen Anfragen

Um die beschriebenen Dienste des AWTS zu ermöglichen, führen wir *Wissensfragmente* als Abstraktionskonzept für das kollektive Wissen aus dem Anfrageprotokoll ein. Ein Fragment formalisiert das mentale Modell einer Gruppe von Datenanalysten zu den verfügbaren Datenquellen über einen bestimmten Zeitraum. Wir verwenden die Bezeichnung *Wissensfragment*, da individuelle mentale Modelle unvollständig sein können, wohingegen die Kombination aller mentalen Modelle in einer Organisation das gesamte Wissen über den Datenbestand bildet. Wie in Abb. 7 zu sehen ist, kapseln die Fragmente verschiedene Wissensmodelle, die wiederum auf Ausschnitten des Anfrageprotokolls basieren. Nachfolgend beschreiben

wir mit Hilfe eines formalen Modells den Lebenszyklus von Fragmenten und erläutern ihre Verwendung durch Datenanalysten.

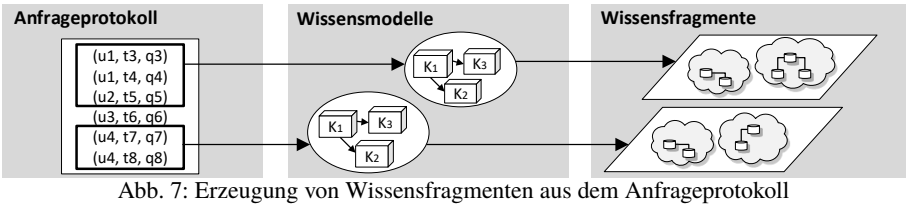


Abb. 7: Erzeugung von Wissensfragmenten aus dem Anfrageprotokoll

2.2.1 Erzeugung und Lebenszyklus von Wissensfragmenten

Wissensfragmente werden gemäß der Wünsche der Datenanalysten erzeugt. Sie bestimmen, welches Wissen in Form von Anfragen berücksichtigt werden soll.
→ **Beispiel:** Initial ist ein Fragment pro Analystenteam sinnvoll, das alle bisherigen Anfragen der Beteiligten beinhaltet. Im Beispielszenario aus Abschnitt 2.1 verwaltet das AWTS zu Beginn daher drei Fragmente.

Extraktion relevanter Anfragen aus dem Anfrageprotokoll Wir modellieren das Anfrageprotokoll L als Menge von Einträgen vom Typ $\mathcal{L}$. Jeder Logeintrag umfasst einen Benutzerschlüssel u vom Typ $\mathcal{U}$, einen Zeitstempel t vom Typ $\mathcal{T}$ und eine Anfrage q vom Typ $\mathcal{Q}$ (Abb. 8 (1)). Nur bestimmte Teile des Anfrageprotokolls L sind für die Datenanalysten interessant. Diese Teile werden mit Funktionen extrahiert, die durch Anwendung einer Menge von Filterprädikaten vom Typ $\mathcal{F}$ auf L eine Teilmenge des Anfrageprotokolls zurückliefern (Abb. 8 (2)). Dabei sind viele Arten von Filterprädikaten und beliebige Extraktionsfunktionen denkbar. Wir stellen exemplarische Definitionen bereit, die für ein AWTS geeignet sind: Jedes Filterprädikat beinhaltet den Benutzer u, dessen Anfragen extrahiert werden sollen, sowie einen Zeitraum für die Extraktion in Form der Zeitstempel t_{start} und t_{end} (Abb. 8 (3)). Unter Verwendung dieser Filterdefinition führen wir eine Funktion `extr` ein, die alle relevanten Anfragen aus dem Anfrageprotokoll L extrahiert (Abb. 8 (4)).
→ **Beispiel:** Im Beispielszenario aus Abschnitt 2.1 werden initial die Filterprädikate F_1 bis F_3 zur Extraktion der relevanten Anfragen verwendet (Abb. 9). Dabei werden jeweils alle Anfragen aller Teammitglieder berücksichtigt. t_α und t_ω zeigen an, dass alle Anfragen über den gesamten Lebenszyklus des AWTS zu berücksichtigen sind. Selbstverständlich ist auch die Angabe fester Zeiträume möglich.

$$L : Set\ \mathcal{L} \quad \text{wobei } \mathcal{L} = (u : \mathcal{U},\ t : \mathcal{T},\ q : \mathcal{Q}) \tag{1}$$
$$\texttt{extr} : Set\ \mathcal{F} \times Set\ \mathcal{L} \rightarrow Set\ \mathcal{L} \tag{2}$$
$$\mathcal{F} = (u : \mathcal{U},\ t_{start} : \mathcal{T},\ t_{end} : \mathcal{T}) \tag{3}$$
$$\texttt{extr}(F, L) = \{l \in L | f \in F.((l.u = f.u) \wedge (f.t_{start} \leq l.t \leq f.t_{end}))\} \tag{4}$$

Abb. 8: Extraktion von relevanten Anfragen

$$F_1 = \{(Alice, t_\alpha, t_\omega), (Bob, t_\alpha, t_\omega)\}$$
$$F_2 = \{(Carol, t_\alpha, t_\omega), (Dan, t_\alpha, t_\omega)\}$$
$$F_3 = \{(Erin, t_\alpha, t_\omega)\}$$

Abb. 9: Beispielhafte Filterprädikate

Erzeugung von Wissensmodellen Aus den extrahierten Teilen des Anfrageprotokolls werden mit Hilfe der vom AWTS mitgelieferten Algorithmen Wissensmodelle erzeugt, die relevante Aspekte der Anfragen formalisieren und somit die Hauptfunktionalität des AWTS realisieren (Abb. 7). Die Algorithmen a_i erzeugen aus einer Teilmenge des Anfrageprotokolls individuelle Datenstrukturen K_i, um das extrahierte Wissen zu repräsentieren (Abb. 10 (5)). Die Indizes i sind Elemente der Indexmenge I, mit deren Hilfe die durch das AWTS bereitgestellten Algorithmen erfasst werden. Jedes Wissensmodell vom Typ $\mathcal{K}$ besteht aus dem Produkt der durch die Algorithmen extrahierten Wissensaspekte (Abb. 10 (6)). Die Funktion $\texttt{createModel}_{L,\texttt{extr},I}$ erzeugt ein Modell, indem die Algorithmen a_i auf einen Teil des Anfrageprotokolls angewendet werden (Abb. 10 (7)).
→ **Beispiel:** Um die im Beispielszenario aus Abschnitt 2.1 beschriebenen Funktionalitäten umzusetzen, bietet das AWTS verschiedene Algorithmen: So kann ein Algorithmus a_{links} in Anfragen vorkommende Verbindungen zwischen Datenquellen in Form eines Graphen aufbereiten. Ein anderer Algorithmus $a_{session}$ kann das Anfrageprotokoll in Sitzungen segmentieren. Ein Algorithmus a_{struct} analysiert die Struktur der angefragten Teile von Datenquellen. Um Ähnlichkeiten zwischen den Vokabularen verschiedener Teams zu erkennen, kann a_{onto} eine Ontologie verwalten. Mithilfe der sich ergebenden Indexmenge I (Abb. 11 (a)) kann das Wissensmodell $\mathcal{K}_1$ für Team 1 erzeugt werden (Abb. 11 (b)).

$$a_i : Set\ \mathcal{L} \rightarrow K_i \quad \text{für } i \in I \tag{5}$$

$$\mathcal{K} = \prod_{i \in I} K_i = (K_{i_1}, ..., K_{i_n}) \tag{6}$$

$$\texttt{createModel}_{L,\texttt{extr},I} : Set\ \mathcal{F} \rightarrow \mathcal{K} \tag{7}$$

$$\texttt{createModel}_{L,\texttt{extr},I}(F) = \prod_{i \in I} a_i\ (\texttt{extr}(F, L))$$

Abb. 10: Erzeugung von Wissensmodellen

$$I = \{links, session, struct, onto\} \tag{a}$$

$$\mathcal{K}_1 = \texttt{createModel}_{L,\texttt{extr},I}(F_1) = \tag{b}$$

$$= (a_{links}\ (\texttt{extr}(F_1, L)), a_{session}\ (\texttt{extr}(F_1, L)),$$

$$a_{struct}\ (\texttt{extr}(F_1, L)), a_{onto}\ (\texttt{extr}(F_1, L))) =$$

$$= (K_{links}, K_{session}, K_{struct}, K_{onto})$$

Abb. 11: Beispielhaftes Wissensmodell

Wissensfragmente Ein Fragment vom Typ $\mathcal{S}$ formalisiert das mentale Modell der kollaborierenden Datenanalysten (Abb. 12 (8)). Dazu wird eine Menge F von Filterprädikaten von einem Analysten spezifiziert, um das Fragment mit den relevanten Anfragen zu initialisieren. Diese Anfragen werden zur Parametrisierung eines Wissensmodells K verwendet. Die Funktion $\texttt{createShard}_{L,\texttt{extr},I}$ nimmt die Menge F von Filterprädikaten vom Typ $\mathcal{F}$, um für ein gegebenes Anfrageprotokoll L, eine gegebene Extraktionsfunktion `extr` und eine gegebene Indexmenge I von Algorithmen das zugehörige Fragment zu erzeugen (Abb. 12 (9)). L, `extr` und I sind für alle Fragmente eines bestimmen AWTS identisch.
→ **Beispiel:** Im Beispielszenario aus Abschnitt 2.1 erzeugen Alice, Carol und Erin gemäß der genannten Filterprädikate jeweils ein Fragment für ihr jeweiliges Team (Abb. 13).

$$\mathcal{S} = (F : Set\ \mathcal{F},\ K : \mathcal{K}) \tag{8}$$

$$\texttt{createShard}_{L,\texttt{extr},I} : Set\ \mathcal{F} \rightarrow \mathcal{S} \tag{9}$$

$$\texttt{createShard}_{L,\texttt{extr},I}(F) = (F, \texttt{createModel}_{L,\texttt{extr},I}(F))$$

Abb. 12: Erzeugung von Wissensfragmenten

$$s_1 = \texttt{createShard}_{L,\texttt{extr},I}(F_1)$$

$$s_2 = \texttt{createShard}_{L,\texttt{extr},I}(F_2)$$

$$s_3 = \texttt{createShard}_{L,\texttt{extr},I}(F_3)$$

Abb. 13: Bsph. Wissensfragmente

Strukturelle Operationen Wissensfragmente sind dynamisch und können sich im Verlauf der Zeit weiterentwickeln, indem neue Anfragen Teil der jeweiligen Wissensmodelle werden.

Um den Datenanalysten mehr Flexibilität zu ermöglichen, stellt das AWTS verschiedene strukturelle Operationen bereit: Zwei Fragmente können verschmolzen werden, um eine weiterführende Kooperation zwischen zwei Analystenteams zu modellieren. Die Funktion $\texttt{merge}_{L,\texttt{extr},I}$ erzeugt dazu mit Hilfe der Filterprädikate zweier Fragmente ein neues Fragment (Abb. 14 (10)). Fragmente können mit $\texttt{expand}_{L,\texttt{extr},I}$ (Abb. 14 (11)) vergrößert oder mit $\texttt{narrow}_{L,\texttt{extr},I}$ (Abb. 14 (12)) verkleinert werden. Dadurch können entweder bestimmte Teile des Anfrageprotokolls hinzugefügt oder ausgeschlossen werden.

$$\texttt{merge}_{L,\texttt{extr},I} : \mathcal{S} \times \mathcal{S} \to \mathcal{S} \quad \texttt{merge}_{L,\texttt{extr},I}(s1, s2) = \texttt{createShard}_{L,\texttt{extr},I}\ (s1.F \cup s2.F) \tag{10}$$

$$\texttt{expand}_{L,\texttt{extr},I} : \mathcal{S} \times \mathcal{F} \to \mathcal{S} \quad \texttt{expand}_{L,\texttt{extr},I}(s, F) = \texttt{createShard}_{L,\texttt{extr},I}\ (s.F \cup F) \tag{11}$$

$$\texttt{narrow}_{L,\texttt{extr},I} : \mathcal{S} \times \mathcal{F} \to \mathcal{S} \quad \texttt{narrow}_{L,\texttt{extr},I}(s, F) = \texttt{createShard}_{L,\texttt{extr},I}\ (s.F - F) \tag{12}$$

Abb. 14: Basisoperationen auf Wissensfragmenten

Vergleichsoperationen Das Erkennen von Ähnlichkeiten zwischen Fragmenten ist ein wichtiger Faktor für deren Weiterentwicklung. Zwei Fragmente sind genau dann ähnlich, wenn ihre Wissensmodelle ähnlich sind. Für den Vergleich von Wissensmodellen wird auf Funktionen $\texttt{sim}K_i$ zurückgegriffen, mit denen jeweils von einem bestimmten Algorithmus a_i aus dem Anfrageprotokoll extrahiertes Wissen verglichen werden kann (Abb. 15 (13)). Wissensmodelle vom Typ $\mathcal{K}$ werden mit Hilfe der Ähnlichkeitsfunktion $\texttt{sim}\mathcal{K}$ verglichen. Mit Hilfe eines Gewichtsvektors w kann der Einfluss einzelner Algorithmen bestimmt werden (Abb. 15 (14)). Die Funktion $\texttt{sim}\mathcal{S}$ bestimmt die Ähnlichkeit zweier Fragmente durch den paarweisen Vergleich ihrer Wissensmodelle (Abb. 15 (15)) gemäß des Gewichtsvektors w. Auf Basis dieses Ähnlichkeitsmaßes generiert das AWTS unter anderem Empfehlungen für strukturelle Operationen.

→ **Beispiel:** Es wird vereinfachend angenommen, dass alle Algorithmen bis auf a_{struct} die Gewichtung 0 erhalten. Aufgrund der strukturellen Ähnlichkeit der Fragmente s_1 und s_2 im Beispielszenario aus Abschnitt 2.1 (s_2 enthält die Hälfte aller Datenquellen von s_1, vgl. Abschnitt 2.1) kann das AWTS empfehlen s_2 mit s_1 zu verschmelzen.

$$\texttt{sim}K_i : K_i \times K_i \to \{x \in \mathbb{R} |\ 0 \leq x \leq 1\} \tag{13}$$

$$\texttt{sim}\mathcal{K} : w \times \mathcal{K} \times \mathcal{K} \to \{x \in \mathbb{R} |\ 0 \leq x \leq 1\} \tag{14}$$

$$\texttt{sim}\mathcal{K}(w, \mathcal{K}_1, \mathcal{K}_2) = \sum_{i \in I} w_i \cdot \texttt{sim}K_i(\mathcal{K}_1.K_i, \mathcal{K}_2.K_i) \quad \text{wobei} \sum_{i \in I} w_i = 1$$

$$\texttt{sim}\mathcal{S} : w \times \mathcal{S} \times \mathcal{S} \to \{x \in \mathbb{R} |\ 0 \leq x \leq 1\} \tag{15}$$

$$\texttt{sim}\mathcal{S}(w, s_1, s_2) = \texttt{sim}\mathcal{K}(w, s_1.K, s_2.K)$$

Abb. 15: Ähnlichkeitsfunktionen für Wissensmodelle und Wissensfragmente

2.2.2 Anfrageverarbeitung

Bei der Benutzung eines AWTS formulieren die Benutzer Anfragen nicht gegen den tatsächlichen Datenbestand, sondern gegen ihr formalisiertes mentales Modell (Abb. 16). Zur Anfrageverarbeitung wird jede Anfrage protokolliert. Das System entscheidet dann, ob die Anfrage nur tatsächlich vorhandene Schemaelemente enthält. Ist dies der Fall, wird die Anfrage normal verarbeitet und die Ergebnisse werden zurückgeliefert. Parallel dazu werden die relevanten Wissensmodelle durch eine Inferenzmaschine ausgewertet. Dadurch

gewonnene Empfehlungen (z.B. ähnliche Datenquellen oder Benutzer mit ähnlichem Informationsbedarf) werden dem Benutzer in einer separaten Oberfläche angezeigt.

Damit die Analysten ihre mentalen Modelle in ihren Anfragen verwenden können, sind auch nicht tatsächlich vorkommende Schemaelemente in Anfragen zulässig. Dazu werden die Wissensmodelle ausgewertet, um die Anfragen durch Modifikationen verarbeitbar zu machen (z.B. Abbildung auf vorhandene Datenquellen). Bei Bedarf wird dazu eine Interaktion mit dem Benutzer angestoßen, damit dieser beispielsweise eine vom System vorgeschlagene Datenquelle auswählen kann, sonst entscheidet das System autonom.
→ **Beispiel:** Für die Empfehlungen im Beispielszenario aus Abschnitt 2.1 greift das AWTS auf die Wissensmodelle der Fragmente s_1, s_2 und s_3 zurück.

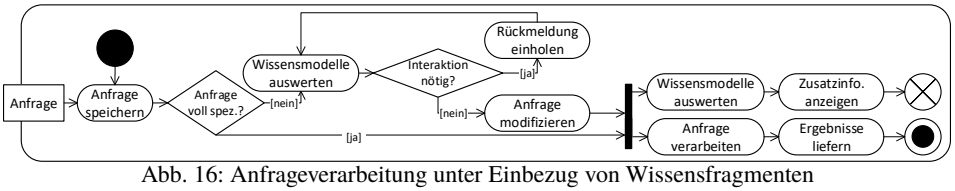


Abb. 16: Anfrageverarbeitung unter Einbezug von Wissensfragmenten

3 Referenzarchitektur

Für die Implementierung eines AWTS schlagen wir eine Client-Server-Architektur vor (Abb. 17). Ein AWTS kommuniziert mit einem vorhandenen Datenmanagementsystem (DMS), um Zugriff auf die verfügbaren Datenquellen zu ermöglichen. Benutzer formulieren Anfragen über die *programmatische Schnittstelle*, die die native Anfrageschnittstelle des DMS kapselt. Etablierte Analyseprozesse können beibehalten werden, da Analysewerkzeuge einfach mit dem AWTS statt mit dem DMS verbunden werden. Die *graphische Benutzeroberfläche* stellt für jede Anfrage relevantes Wissen dar und präsentiert Anfragemodifikationen. Darüber hinaus können hiermit Lebenszyklusoperationen auf den Fragmenten durchgeführt werden. Eingehende Anfragen werden protokolliert und nach eventuellen Modifikationen an das DMS weitergeleitet. Die *Fragmentverwaltung* ist für den Lebenszyklus aller Fragmente verantwortlich und erzeugt die Wissensmodelle. Zudem überwacht sie das Anfrageprotokoll, um bei Bedarf die Wissensmodelle zu aktualisieren. Die Komponente zum *Wissenstransfer* stellt relevantes Wissen bereit und stößt Anfragemodifikationen an. Dazu werden die Wissensmodelle mit Hilfe der *Inferenzmaschine* ausgewertet. Sie ist auch für die Überwachung der Fragmente zuständig, um Empfehlungen für Lebenszyklusoperationen abzuleiten.

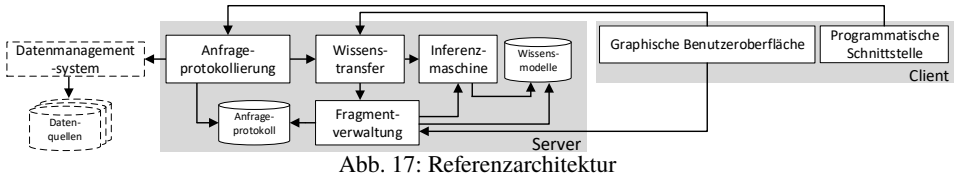


Abb. 17: Referenzarchitektur

Für weitere Untersuchungen implementieren wir aktuell auf Basis von *Apache Drill* ein AWTS, das den Zugriff auf verschiedene Typen von Datenquellen mittels SQL ermöglicht. Dazu stellen wir unter anderem Algorithmen für Anfrageähnlichkeit, Sitzungserkennung, Schemazusammenführung und Gewichtung von Verbundattributen bereit.

4 Evaluationsansatz

Die Evaluation unserer Ergebnisse erfolgt zweistufig. Zum einen wird überprüft, ob unser Ansatz einen konkreten Mehrwert für Datenanalysten bietet und ihnen die Beantwortung analytischer Fragestellungen erleichtert. Zum anderen wird untersucht, ob die Leistungsfähigkeit unserer Referenzimplementierung für analytische ad-hoc-Anfragen ausreichend und somit eine interaktive Verwendung des Systems möglich ist.

Der praktische Nutzen unseres Ansatzes wird mit einer Benutzerstudie in einer konkreten Anwendungsdomäne überprüft. In einem ersten Schritt wird dazu die Korrektheit der Wissensmodelle evaluiert. Auf Basis von realen Benutzeranfragen werden Modelle erzeugt. Die Benutzer bewerten anschließend, inwiefern diese Modelle ihre Intentionen widerspiegeln. Darüber hinaus werden die zur Modellerstellung notwendige Anzahl von Anfragen und die Güte der Ähnlichkeitsfunktionen zum Modellvergleich untersucht. In einem zweiten Schritt erhalten zwei Benutzergruppen die Aufgabe, konkrete Fragestellungen mit Hilfe vorgegebener Datenquellen zu beantworten. Beide Gruppen verwenden das AWTS, um auf die Daten über eine einheitliche Schnittstelle zuzugreifen. Aber nur eine Gruppe wird vom System durch Empfehlungen auf Basis von Wissensmodellen unterstützt. Durch einen Vergleich der Arbeitsgeschwindigkeit und der Ergebnisqualität beider Benutzergruppen wird untersucht, ob die Funktionen zum Anfrage-getriebenen Wissenstransfer die Beantwortung analytischer Fragestellungen erleichtern.

Zur Bestimmung der Leistungsfähigkeit unserer Referenzimplementierung werden Rechenzeit und Speicherverbrauch für die Modellerzeugung, die Modellaktualisierung, die Modellauswertung und den Modellvergleich gemessen. Zudem werden die Antwortzeiten des Gesamtsystems bei der Verarbeitung von ad-hoc-Anfragen untersucht.

Bei der Evaluation der Leistungsfähigkeit kann unter anderem auf synthetische Anfrageprotokolle zurückgegriffen werden, die von einem Anfragegenerator erzeugt werden. Zur Beurteilung der Nützlichkeit unseres Ansatzes sind reale Anfrageprotokolle erforderlich. Neben den Anfragen, die im Rahmen der Benutzerstudie anfallen, können zusätzlich auch Anfragen aus bereits eingesetzten Datenmanagementsystemen herangezogen werden.

5 Verwandte Arbeiten

Unser Ansatz ist mit Dataspace-Systemen [FHM05] verwandt, die auf Basis von Benutzerrückmeldungen die verwalteten Daten inkrementell an die Erwartungen der Benutzer anpassen. Existierende Systeme berücksichtigen jedoch keine Szenarien, in denen Benutzergruppen mit heterogenen Erwartungen mit einer gemeinsamen Menge von Datenquellen arbeiten. Explizite Benutzerrückmeldungen waren bereits Forschungsgegenstand, Wissen aus analytischen Anfragen bleibt jedoch noch ungenutzt [Be13]. Khoussainova et al. fordern die Entwicklung von Systemen zur Verwaltung von Anfragen [Kh09]. Teile ihrer Vorschläge wurden von verschiedenen Forschungsprojekten zur Auswertung einzelner Aspekte von Anfrageprotokollen aufgegriffen. Einen Überblick über diese Ansätze liefern unsere Vorarbeiten [Wa16, Sc16].

Mentale Modelle von Analysten sind nicht immer kongruent zum vorhandenen Datenbestand. Damit sie ihre mentalen Modelle dennoch in Anfragen verwenden können, ermöglichen

einige Ansätze die Referenzierung unbekannter Schemaelemente in Anfragen [Eb15, LPJ14]. Komplexe temporale und soziale Zusammenhänge, die aus dem Anfrageprotokoll extrahiert werden können, finden jedoch keine Berücksichtigung.

Unser Ansatz unterscheidet sich von der Empfehlung ganzer Anfragen [Ei14] und der automatischen Vervollständigung von Anfragen [Kh10], weil wir den Analysten das Formulieren vollständiger Anfragen unter Verwendung ihrer mentalen Modelle ermöglichen. Ziel ist es, den Datenbestand an die mentalen Modelle der Benutzer anzupassen und nicht die Benutzer zur Anpassung ihrer mentalen Modelle zu zwingen.

6 Zusammenfassung

Um von der Verfügbarkeit von heterogenen Datenquellen zu profitieren, muss deren effiziente Nutzung sichergestellt werden. Wir schlagen *Anfrage-getriebene Wissenstransfersysteme (AWTS)* als Ansatz zur inkrementellen Datenintegration vor. Mit einem AWTS können Datenanalysten stilles Wissen über Datenquellen ohne manuellen Dokumentationsaufwand externalisieren. Sie lernen, wie andere Analysten die Datenquellen verwenden und können die Semantik relevanter Datenquellen einfacher verstehen. Ein AWTS stellt neuartige Funktionalitäten bereit, die Datenanalyseprozesse unterstützen können. Gemäß unserer Referenzarchitektur erfordert der Einsatz eines AWTS keine Änderung von etablierten Analyseprozessen, da bestehende Analysewerkzeuge weiterhin verwendet werden können.

Literaturverzeichnis

[Be13] Belhajjame, K.; Paton, N. W.; Embury, S. M.; Fernandes, A. A.; Hedeler, C.: Incrementally Improving Dataspaces Based on User Feedback. Inf. Syst., 38(5):656–687, Juli 2013.

[Eb15] Eberius, J.; Thiele, M.; Braunschweig, K.; Lehner, W.: DrillBeyond: Processing Multi-result Open World SQL Queries. In: SSDBM '15. ACM, S. 16:1–16:12, 2015.

[Ei14] Eirinaki, M.; Abraham, S.; Polyzotis, N.; Shaikh, N.: QueRIE: Collaborative Database Exploration. IEEE Trans. on Knowledge and Data Eng., 26(7):1778–1790, Juli 2014.

[FHM05] Franklin, M.; Halevy, A.; Maier, D.: From Databases to Dataspaces: A New Abstraction for Information Management. SIGMOD Rec., 34(4):27–33, Dezember 2005.

[Kh09] Khoussainova, N.; Balazinska, M.; Gatterbauer, W.; Kwon, Y.; Suciu, D.: A case for a collaborative query management system. In: CIDR. 2009.

[Kh10] Khoussainova, N.; Kwon, Y.; Balazinska, M.; Suciu, D.: SnipSuggest: Context-aware Autocompletion for SQL. Proc. VLDB Endow., 4(1):22–33, Oktober 2010.

[LPJ14] Li, F.; Pan, T.; Jagadish, H. V.: Schema-free SQL. In: SIGMOD'14. ACM, S. 1051–1062, 2014.

[NK98] Nonaka, I; Konno, N.: The concept of "ba": Building a foundation for knowledge creation. California mgmt. review, 40(3):40–54, 1998.

[Sa11] Saeed, M.; Villarroel, M.; Reisner, A. T.; Clifford, G.; Lehman, L. et al.: Multiparameter Intelligent Monitoring in Intensive Care II (MIMIC-II): a public-access intensive care unit database. Critical care medicine, 39(5):952, 2011.

[Sc16] Schwab, P. K.; Wahl, A. M; Lenz, R.; Meyer-Wegener, K.: Query-driven Data Integration (Short Paper). In: FG-DB'16. GI, 2016.

[St15] Stonebraker, M.: The Case for Polystores. Juli 2015. `http://wp.sigmod.org/?p=1629`, abgerufen: 01.09.2016.

[Wa16] Wahl, Andreas M: A minimally-intrusive approach for query-driven data integration systems. In: ICDEW'16. IEEE, S. 231–235, 2016.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 175

Metadata Management for Data Integration in Medical Sciences

- Experiences from the LIFE Study -

Toralf Kirsten[1,2] Alexander Kiel[2] Mathias Rühle[2] Jonas Wagner[2]

Abstract: Clinical and epidemiological studies are commonly used in medical sciences. They typically collect data by using different input forms and information systems. Metadata describing input forms, database schemas and input systems are used for data integration but are typically distributed over different software tools; each uses portions of metadata, such as for loading (ETL), data presentation and analysis. In this paper, we describe an approach managing metadata centrally and consistently in a dedicated Metadata Repository (MDR). Metadata can be provided to different tools. Moreover, the MDR includes a matching component creating schema mappings as a prerequisite to integrate captured medical data. We describe the approach, the MDR infrastructure and provide algorithms for creating schema mappings. Finally, we show selected evaluation results. The MDR is fully operational and used to integrate data from a multitude of input forms and systems in the epidemiological study *LIFE*.

Keywords: Data Integration, Schema Matching, Schema Merging, Metadata Repository

1 Introduction

Clinical and epidemiological studies and other health related surveys are often used in evidence-based medical sciences. Clinical studies investigate specific biological and medical phenomena and their implications from the clinical point of view. For instance, they evaluate new therapy procedures according to specific diseases, test the application of new drugs or drug doses under various circumstances by comparing the results (e.g., survival rate for cancer) to previous therapy procedures accepted in the medical community. Epidemiological studies and other health surveys usually study the development of prevalences (ratio of infected to all persons) for specific diseases and health imbalances by determining a population in a geographical region of interest. The increasing prevalence rate of obesity in industrial countries (in contrast to development countries) is a finding of such studies; the goal is to find specific explanations for this increase, e.g., changes of life style or nutrition habits, in order to show trends and impacts (also for health care policy). In both, clinical and epidemiological studies, it is quite common to determine a set of participants by different investigations, so called assessments, in which data are captured about the participant and need to be integrated before a comprehensive data analysis can start.

LIFE [Qu12, Lo15] is a large epidemiological study at the University of Leipzig in the described context. The goal of *LIFE* is to determine the causes of common civilization

[1] Interdisciplinary Centre for Bioinformatics, Univ. of Leipzig, Härtelstr. 16-18, 04107 Leipzig
[2] LIFE Research Centre for Civilization Diseases, Univ. of Leipzig, Philipp-Rosenthal-Str. 27, 04103 Leipzig
tkirsten@izbi.uni-leipzig.de, akiel,mruehle,jwagner@life.uni-leipzig.de

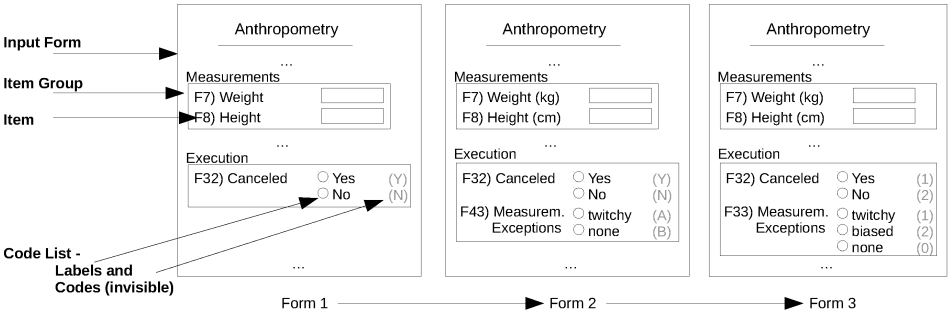


Fig. 1: Exemplified Evolution of Assessment "Anthropometry" with multiple Input Forms

diseases including adiposity (obesity), depression, dementia, diabetes melitus, allergy, cardiovascular disease and heart attack. By now, more than 23,000 participants who mostly are Leipzig (Germany) inhabitants have been examined. Each selected and invited person is associated to a special investigation program consisting of different assessment types including questionnaires, interviews, physical examinations and sample extraction (e.g., blood and urine samples). An assessment specifies how (the procedure) and which data are captured. Each assessment is implemented by a number of input forms; each form by a specfic input system. Such input systems are web-based systems, desktop systems, or spreadsheet systems allowing online and subsequent data input which is often manually and directly carried out by the ambulance staff or the participant. Furthermore, medical devices producing data by a measurement process (during a physical examination) often need a special "reading" process and pre-processing to derive and extract data from proprietary formats, such as images, 3D models etc.

Fig. 1 (left) shows a portion of a single input form implementing the Anthropometry assessment. The form consists of different data items (e.g., questions of questionnaires), such as body height and weight, for which data can be captured. Multiple items can be grouped into item groups, i.e., they form a group of input fields on the input form. In *LIFE*, input forms and other data sources can change over time, a fact that makes the integration effort much more challenging. There are several reasons for that. Misspellings in data items text and wrong or missing validation rules of input forms should be corrected. Other input forms need to be adapted to meet specific requirements at the investigation time or extended research questions, e.g., by adding or deleting data items to/from an input form. These changes do not necessarily result in structural changes, in particular correction of misspellings. However, the input systems used in *LIFE* allow to change every input form until it is activated for data input. Further changes result in designing and activating a new input form while the previous form can be deactivated to prevent further data capturing with this form. Each input form is associated with a data table (database or spreadsheet) in which the captured data are stored. Using multiple input forms per assessment results in multiple data tables.

Fig. 1 shows the evolution of the assessment Anthropometry using three input forms. While the text of data items weight and height changes from form 1 to 2, the item identifier F43 moves to F33 due to rearrangements of questions in version 3. New items are introduced in form 2 and 3. Moreover, the codes of the answer lists (so called code lists in trial management) are numeric in version 3; they were alphabetic in forms 1 and 2. Similarly, the schema of generated data heavily depends on the used software (and version), such that schema changes are the normal case. A data analysis, however, should run on integrated data for each assessment instead of multiple input forms. In *LIFE* , there are currently more than 900 assessments with more than 52,000 items (in total) which are associated with more then 1,700 input forms (approx. 120,000 items). A manual integration connecting input forms with assessment is, therefore, very resource-intensive and error-prone. To reduce this manual effort we designed and implemented an integration approach which we introduce in this paper. In particular, we make the following contributions.

- Our integration approach uses current schema matching and merging techniques to generate mappings to harmonize the schema of input forms and other data sources as well as code lists of data items. The advantage is twofold. First, the target schema can be automatically derived and generated. Secondly, data can be transformed and transfered by an automatic process using the generated mappings. We introduce the applied matching techniques and provide algorithms.
- All metadata is collected and managed by a separate Metadata Repository (MDR) which is implemented by a service-based infrastructure. The MDR is fully operational and is used for several years in *LIFE* for data integration. We introduce the system architecture and show an overview of its database schema.
- Taking the available and validated mapping data from the productive instance, we evaluate the quality of our mapping-based approach and discuss the results.

The rest of the paper is organized as follows. In Sect. 2, we introduce some basic models and definitions that we use throughout the paper. Sect. 3 explains the schema matching process we have implemented and shows algorithms allowing to find corresponding schema elements. We sketch the match infrastructure in Sect. 4, show selected evaluation results in Sect. 5 and discuss related work in Sect. 6 before we conclude in Sect. 7.

2 Preliminaries and Models

An input form $F = (G, I, R_F)$ consists of a non-empty set of items I (e.g., questions in questionnaires and interviews). Items are organized as ordered set (i.e., list) into item groups G. The input form structure $R_F \subset G \times I$ specifies which group contains which items. Therefore, an input form is organized as tree with F as root, groups as inner nodes and items as leaves. Input forms as well as item groups and items are described by a non-empty set of attributes. Input forms and item groups normally have a title, whereas items are primarily described by an item identifier and a description, i.e., the question text in a questionnaire and the measurement parameter name in physical investigations. Further item attributes depend on the utilized input system, such as the item representation on the web page (radio button, check box, short text field etc.). Furthermore, categorial data items are associated

with a predefined list of instances (i.e., categories subsumed as code list) to restrict the set of answers. Each code list element is represented by a code and a label. While the label is shown on the input web-page or used as label when analysis results are presented, the code is directly represented within data and, hence, only "internally" used.

Throughout the paper, we use the term *input form* and its above definition for forms that are interactively used by study participants and study staff to manually capture data but also for forms which are "automatically filled", e.g., when devices produce data. Forms of the latter case only exist "virtually"; they are used to specify metadata since database schemas of device software installations or data exports (e.g., spreadsheets) typically don't provide item descriptions and code lists. Moreover, we add the input system IS and a version number v (linear versioning schema per input system) as index to each input form $F_{IS,v}$ allowing us to differentiate and to address a single input form of an assessment. Note, there can be multiple input forms simultaneously used at the same time, e.g., input forms of the same or different input systems.

Every input form $F_{IS,v}$ is associated to a schema $S_{IS,v} = (E, R_S)$ which is used by an input system to internally store and manage captured data of the input form $F_{IS,v}$. In this paper, we focus on relational schemas consisting of tables and columns (generalized as set of schema elements E and their relationships $R_S \subset E \times E$). We assume that every input form can be represented as single table or single denormalized view on multiple tables. In cases in which data are not represented in a relational schema, we apply a transformation step and import the data into a relational schema. The mapping $M_{F_{IS,v},S_{IS,v}} = (F_{IS,v}, S_{IS,v}, C_{F,S})$ explicitly represents the interrelation between $F_{IS,v}$ and $S_{IS,v}$; the set $C_{F,S}$ consists of correspondences specifying which schema element of $S_{IS,v}$ is associated to which data item of input form $F_{IS,v}$. Note, an input form can contain data items which have no correspondence to any schema element; typically, they show derived data, e.g., special text paragraphs or computations (e.g., body mass index) that are dynamically created based on data of other data items for which data have been captured. Furthermore, there are data items which are closely related. For instance, there are questions in questionnaires and interviews associated with a multiple choice behavior. There, each element of the code list is represented by a check box on the input form; each check box can be enabled independently from the other. Therefore, the relational schema contains an element (column) for each code list element (check box). Mostly, the mapping $M_{F_{IS,v},S_{IS,v}}$ is inherently generated by the input systems we use in *LIFE*; only for spreadsheets and exported data files from medical devices we generate a mapping manually by creating (virtual) input forms for a specified relational schema after importing data into the database. For simplicity, we skip the input system as index in the rest of the paper and denote an input form F (source schema S) in version v by F_v (S_v) instead of $F_{IS,v}$ ($S_{IS,v}$).

The goal of the schema matching process is to find semantic meaningful correspondences between elements of two distinct schemas, say S_i and S_j. The resulting correspondence set C_{S_i,S_j} builds a mapping $M_{S_i,S_j} = (S_i, S_j, C_S)$. A special schema is the target schema T. In our case, T represents the schema of the so called research database managing data from all input forms in a harmonized manner. Hence, we are finally interested in schema mappings $M_{S_i,T}$ between schemas of input forms S_i and the target schema T.

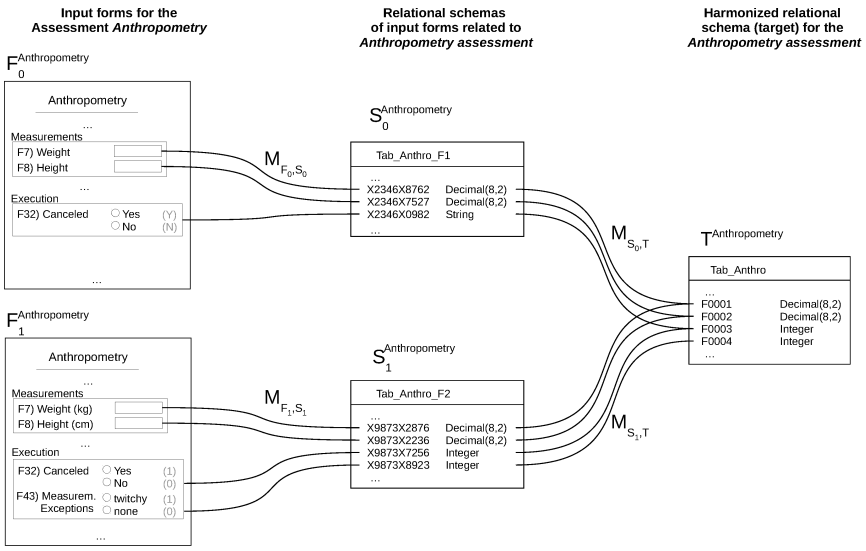


Fig. 2: Input Form, Schemas and their interrelating Mappings

A single correspondence $c(e_{S_i}, e_{S_j}, Exp) \in C_S$ associates the schema elements $e_{S_i} \in S_i$ and $e_{S_j} \in S_j$ using a transformation expression Exp, i.e., $e_{S_j} = Exp(e_{S_i})$. The transformation expression is implemented by database functions and used by the ETL (extraction, transformation, loading) process when the data of sources are transferred to the target. Transformation expressions are typically data type conversions, e.g., to convert text values into numbers or dates. We also use transformation expressions when code list values need to be harmonized across input forms, i.e., we apply an code list mapping associating each element of the code list of a source schema element with an element of the code list of the corresponding target schema element. Moreover, there is also an identity expression such that $e_{S_j} = Exp(e_{S_i}) = e_{S_i}$.

Fig. 2 shows an example for the described models and their relationships. There are two input forms for the interview assessment "Anthropometry", $F_0^{\text{Anthropometry}}$ and $F_1^{\text{Anthropometry}}$, respectively. Both input forms consist of a set of data items for which a selected portion is shown. The input form $F_0^{\text{Anthropometry}}$ ($F_1^{\text{Anthropometry}}$) corresponds to schema $S_0^{\text{Anthropometry}}$ ($S_1^{\text{Anthropometry}}$). The schema is mostly created by the input systems taking the user specifications of the input form into account. Hence, the user influence on naming of schema elements, their data types and utilized constraints depends on the utilized input system; while some systems take user input into account other input systems are very restrictive.

There are three changes between $F_0^{\text{Anthropometry}}$ and $F_1^{\text{Anthropometry}}$. Firstly, item descriptions of weight (F7) and height (F8) have been changed. Secondly, the input form $F_1^{\text{Anthropometry}}$ extends the input form $F_0^{\text{Anthropometry}}$ by a new data item (F43). Finally, both input forms contain a data item F32 that is associated with a code list. While in $F_0^{\text{Anthropometry}}$ the code list consists of elements *yes* (decoded by Y) and *no* (N), the code list in $F_1^{\text{Anthropometry}}$ utilizes

numeric codes 0 (no) and 1 (yes) instead of alphabetic codes. The changed item descriptions (for items weight and height) have no effect on the schema $S_1^{\text{Anthropometry}}$. However, schema $S_1^{\text{Anthropometry}}$ has an additional schema element storing values of the new data item F43. The changed code lists also result in a numeric data type for data item F32 in comparison to $S_0^{\text{Anthropometry}}$ (String). Finally, both schemas need to be associated to the target schema $T^{\text{Anthropometry}}$ to harmonize data within the ETL process. For this reason, we generate and use mappings $M_{S_0^{\text{Anthropometry}},T^{\text{Anthropometry}}}$ and $M_{S_1^{\text{Anthropometry}},T^{\text{Anthropometry}}}$, respectively.

3 Matching and Merging Schemas

We firstly describe processes generating and verifying schema mappings. Finally, we introduce the utilized matching techniques and algorithms.

3.1 The overall Schema Matching Process

The goal of our schema matching process is to find correspondences between the elements of a source schema and the target schema. We utilize the generated mappings to transfer the data from each source to the target. Moreover, we use schema mappings to answer data provenance questions, i.e., to exactly specify from which input form (and item) the data come from when looking on the target schema and data. However, schema matching is challenging for two reasons. First, we don't use a predefined target schema for each assessment; defining the target schema apriori and adapt this schema with potentially each schema correnssponding to an input form is too resource intensive. Secondly, most source schemas are generated by input systems. In some cases the form creator has no influence on the naming of schema elements; names are concatenated internal identifiers (e.g., form id + group id + item id). Hence, creating schema mappings by taking (only) schemas as input is not sufficient. Addressing these two challenges, our approach consists of two steps. The first step generates the target schema taking a selected input form into account. The second step creates schema mappings between each source schema and the generated target schema by transforming the schema matching task to a form matching task producing mappings between different input forms belonging to the same assessment. Due to the duality between forms and schemas, the resulting form mappings can be used to derive required schema mappings. Fig. 3 shows both processes, the generation of the target schema for a selected input form and the creation of schema mappings for further input forms of the same assessment.

The mapping process starts when a new input form is created and utilized to capture data. At first, metadata about the input form and its corresponding schema is imported into the centralized Metadata Repository (MDR, see also Sect. 4). Then, the schema element *table* is associated with the input form metadata, i.e., name and description of the input form are related to the corresponding *table* element of the relational schema. Columns of this table are associated to data items of the input form and, therefore, each column is related to an item text (i.e., question/parameter text). To keep this import process simple and to reduce the implementation effort, we import the source schema and its annotation directly from the input systems managing input forms. For all other cases, e.g., data exports of

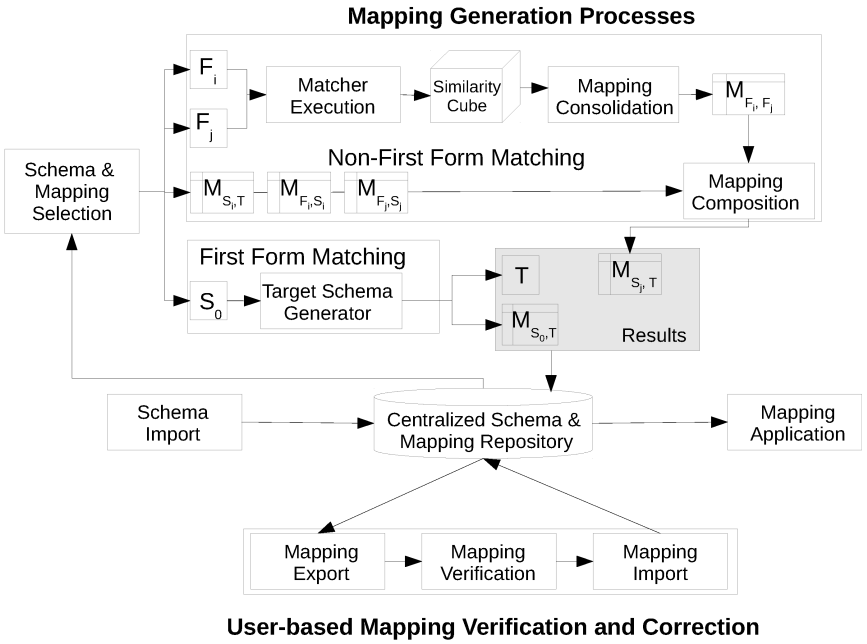


Fig. 3: Mapping Generation and Verification Processes

medical devices, this metadata are specified in a spreadsheet by responsible users that can be imported into the MDR.

By selecting an input form for which the corresponding source schema and additional form metadata are available in the MDR, we can derive the target schema for an assessment (part "first form mapping" in Fig. 3). Typically, we select the first input form that have been chronologically created and used to load and access harmonized data to/of the target database. The constructed target schema consists of a column for each source column (1:1 mapping). The schema mapping is inherently created and persisted within the MDR. The names of the target schema (table and column name) are automatically created following a global notation. The user can later (in the verification process) specify an alias for each column to associate a semantic meaningful name. According to the source schema, the generated correspondences of the schema mapping consists of a transformation expression. There is a predefined set of conversion functions to transform values w.r.t. their data type since the used input systems in *LIFE* mostly utilize VARCHAR or TEXT (CLOB) as standard data type which are typically not the preferred type in the target schema. Therefore, data values are parsed during the process to recognize the data type (numeric or date). Since most items (questions) in questionnaires and interviews utilize predefined code lists which have mostly not more than 10 answers, this task is not very resource intensive. Only for open questions (no predefined code list) we parse the first 100 values. Special value transformations, such as from feet to centimeter, are not recognized; they can be added and set on demand by manual interaction in the verification process.

Note, that the target finally contains tables for each assessment. There is (currently) no target table that is associated with input forms of multiple assessments. The reason for this is that the structure should be harmonized first to overcome the heterogeneity of different input forms of potentially multiple input systems. In additional steps, the target tables can then be joined on demand to meet the data requirements of diverse analysis projects. That allows us to automatically derive and adopt the target schema from the source schemas during the schema matching process instead of creating it manually and a priori.

Once the target schema has been created for a selected input form, we generate mappings between source schema and the generated target schema for all further input forms at the time they are created and productively used (part "non-first form mapping" in Fig. 3). Since names of schema elements are typically generated by input systems, they can't be used to successfully generate schema mappings. Therefore, the central idea is to move the schema match problem to a matching between input forms, firstly. We select two input forms F_i and F_j as input for this match process. For one of these input forms, say F_i, a schema mapping $M_{S_i,T}$ for its corresponding source schema S_i already exists whereas for S_j (corresponding to F_j) there is no mapping $M_{S_j,T}$ available. To derive the mapping $M_{S_j,T}$, we generate the mapping M_{F_i,F_j} and compose the result with the mappings M_{F_i,S_i} and M_{F_j,S_j} between forms and its corresponding schema. Hence, the composition generates the intermediate mapping $M'_{S_j,T}$ by

$$M'_{S_j,T} := (M_{F_j,S_j})^{-1} \circ (M_{F_i,F_j})^{-1} \circ M_{F_i,S_i} \circ M_{S_i,T}$$

The inverse of a given mapping $M_{X,Y}$ is defined as $(M_{X,Y})^{-1} = ((M_{Y,X})^{-1})^{-1} = M_{Y,X}$. Deleted items, i.e., all items present in F_i but not in F_j, have no correspondence in M_{F_i,F_j}. Similarly, added items, i.e., all items present in F_j but not in F_i, also have no correspondence in M_{F_i,F_j}. Only the latter case is of interest for $M'_{S_j,T}$ because there are potentially schema elements in S_j (new data items) with missing correspondences. Therefore, we generate a new schema element in T and inherently new correspondences for each schema element in S_j which has no counterpart in T as defined by $M'_{S_j,T}$. This generated correspondence set is added to $M'_{S_j,T}$ and builds the final $M_{S_j,T}$. Note, we do not differentiate between new items in F_j and those for which the match process does not bring out a correspondence in M_{F_i,F_j}, e.g., due to low similarity of questions texts.

This approach has several advantages. We reuse the mapping $M_{S_i,T}$. This mapping is manualy checked (see below) and, thus, has a good quality. The mappings M_{F_i,S_i} and M_{F_j,S_j} are inherently given when a new input form together with its schema is imported into the MDR. Hence, the quality of $M'_{S_j,T}$ is mainly driven by the quality of the computed mapping M_{F_i,F_j}. Moreover, transformation expressions specified for schema correspondences of $M_{S_i,T}$ can be reused. For example, a schema correspondence associates a text column of S_i containing date values that are transformed into dates using a special conversion expression, then we expect that the schema element in S_j for the same target item comprises also date values and the conversion expression can be applied again.

In a separate process (part "user-based mapping verification and correction" in Fig. 3), we allow users to modify the automatically generated mapping. The user feedback is important

to correct the current mapping but also has implications to future mappings since it is reused in the next match process. We allow the user to change the target schema but also the mapping itself. The former includes data type changes and aliasing of target columns. We also allow the user to modify the code and label of predefined answers since we have experienced different codes and labels of such answers over different input form versions. Mapping modifications occur when the user replaces the automatically determined transformation expression or decide to split a target column, e.g., when the predefined answer set for items using the same (question) text/description in two consecutive input form versions differ significantly. This decision can not be automatically derived but needs human interaction.

We currently use a spreadsheet to verify and modify a mapping. The spreadsheet is protected in some area to show as much information as possible and to modify as much information as necessary. Once a mapping is verified it can be imported and updated in the central MDR. This verification process on a single mapping can be iteratively executed to continually improve schema mappings. In the near future, we will work on a web-based user interface allowing selected users to verify and modify mappings online.

3.2 Schema Matching Techniques and Algorithms

In the following, we introduce selected algorithms computing the mapping for all input forms $F_0 < F_i \leq F_n$ of non-first input forms; the generation of the schema mapping $M_{S_0,T}$ for the first input form F_0 is simple as described in the previous subsection. The Algorithm 1 shows the overall matching process generating mappings $M_{S_i,T}$ for non-first forms F_i ($i > 0$); this process has been introduced in the previous subsection. Algorithm 2 and Algorithm 3 show the process of aligning two selected input forms F_{i-1} and F_i in more detail. We use a blocking strategy in the match process determining the form mapping construction. The overall idea of blocking is to limit the search space and, thus, to reduce the number of

Algorithm 1 Non-First-Form Match

Require: Input Forms F_{i-1}, F_i, mappings $M_{F_{i-1},S_{i-1}}$, M_{F_i,S_i}, target schema T, schema mapping $M_{S_{i-1},T}$
Ensure: Schema mapping $M_{S_i,T}$

1: M_{F_{i-1},F_i} := match-form(F_{i-1}, F_i)
2: $M_{S_i,F_{i-1}}$:= compose(inverse(M_{F_i,S_i}), inverse(M_{F_{i-1},F_i}))
3: $M_{S_i,S_{i-1}}$:= compose($M_{S_i,F_{i-1}}$, $M_{F_{i-1},S_{i-1}}$)
4: $M'_{S_i,T}$:= compose($M_{S_i,S_{i-1}}$, $M_{S_{i-1},T}$)
5: $S_i^{\text{non-matched}}$:= $S_i \setminus$ domain($M'_{S_i,T}$)
6: create schema mapping $M''_{S_i^{\text{non-matched}},T}$ with empty correspondence set
7: **for** $e_{S_i} \in S_i^{\text{non-matched}}$ **do**
8: create new schema element (column) e_t in target schema T
9: create new correspondence $c(e_{S_i}, e_t, 1) \rightarrow M''_{S_i^{\text{non-matched}},T}$
10: **end for**
11: $M_{S_i,T}$:= union($M'_{S_i,T}$, $M''_{S_i^{\text{non-matched}},T}$)
12: **return** $M_{S_i,T}$

Algorithm 2 Match-Form

Require: Input Forms F_{i-1}, F_i, threshold t
Ensure: Form mapping M_{F_{i-1},F_i}
1: **for** $g_{F_{i-1}} \in G_{F_{i-1}}$ **do**
2: **for** $g_{F_i} \in G_{F_i}$ **do**
3: **if** $g_{F_{i-1}}.title = g_{F_i}.title$ **then**
4: M_{F_{i-1},F_i} := union(M_{F_{i-1},F_i}, match-block($g_{F_{i-1}}$, g_{F_i}, t)
5: **end if**
6: **end for**
7: **end for**
8: **return** filterBest(M_{F_{i-1},F_i})

element comparisons. A block consists of all items belonging to a predefined item group. Each block is identified by the group title; there are no two groups within a single input form using the same group title. We also expect that the group title does not change over time and is the same in forms of different input systems. The reason for this is that new input forms are mainly created by modifying the most recent input form. The applied change operations consist in *add* (e.g., adding new items), *delete* (e.g., deletion of existing items), and *change*. The latter operation is mostly applied to re-arrange items in a single item group, e.g., by changing the rank order of items or by manipulating the item text to correct misspellings or misunderstandings. Moreover, using the group title as block identifier simplifies the block construction. Therefore, a separate block construction in a pre-processing can be avoided saving runtime and main memory consumption.

The data items of two blocks are only compared when their corresponding block identifier is identical. The reason for this is twofold. Firstly, in some cases there are a lot of very similar group titles within a single input form, such as *questions 1*, *questions 2*, ... or *medication 1*, *medication 2*, ... etc. which makes it challenging to find the most similar group. Secondly, some input forms contain recurring items, i.e., a defined list of items (questions) occurring on several pages. In such cases, the participant gives information to an

Algorithm 3 Match-Block

Require: Blocks $g_{F_{i-1}}$, g_{F_i}, threshold t
Ensure: Block-based schema mapping $M_{g_{F_{i-1}},g_{F_i}}$
1: create $M_{g_{F_{i-1}},g_{F_i}}$ with empty correspodence set
2: **for** $item_{F_{i-1}} \in g_{F_{i-1}}.items$ **do**
3: **for** $item_{F_i} \in g_{F_i}.items$ **do**
4: $s := similarity(item_{F_{i-1}}, item_{F_i}, t)$
5: **if** $s \geq t$ **then**
6: create correspondence $c(item_{F_{i-1}}, item_{F_i}, s) \rightarrow M_{g_{F_{i-1}},g_{F_i}}$
7: **end if**
8: **end for**
9: **end for**
10: **return** $M_{g_{F_{i-1}},g_{F_i}}$

Algorithm 4 filterBest

Require: Form mapping M_{F_{i-1},F_i}
Ensure: Filtered form mapping $M_{F_i,F_{i-1}}$
1: $Set^{Item}_{Domain} := \emptyset$, $Set^{Item}_{Range} := \emptyset$
2: C_{sorted} := sort C of $M_{F_i,F_{i-1}}$ order by descending similarity
3: **for** $c \in C_{sorted}$ **do**
4: **if** domain(c)$\in Set^{Item}_{Domain}$ OR range(c) $\in Set^{Item}_{Range}$ **then**
5: remove c from $M_{F_i,F_{i-1}}$
6: **else**
7: domain(c) $\rightarrow Set^{Item}_{Domain}$
8: range(c) $\rightarrow Set^{Item}_{range}$
9: **end if**
10: **end for**
11: **return** M_{F_{i-1},F_i}

undefined number of medications, diseases in the past, children etc.; mostly this undefined number is a priori limited by the number of groups that the form designer created. However, such group replications result in multiple correspondences between data items and it is hard to automatically decide which element of form F_{i-1} refers to which element in form F_i when the group information is ignored.

We utilize two specific matchers to decide whether two data items are equivalent. Both matchers take two items as input and return the calculated similarity between them. The first matcher computes the trigram similarity of concatenated item code and text (e.g., question text). The second matcher compares two items by the overlap of their corresponding pre-defined code lists. The reason is that two items using similar question text but mainly differ in their code lists should not be merged since a merge of two items also implies the merge of their code lists. Finally, a correspondence between both items is created and added to the resulting form mapping when the averaged similarity of both matcher exceeds a given threshold t.

The match process creates mapping portions for each block-wise comparison (see Algorithm 2) which are added to the intermediate mapping M'_{F_{i-1},F_i}. This mapping typically contains multiple correspondences for a single schema element of F_{i-1} and F_i since the correspondence creation is only limited by the given threshold. Currently, we utilize a threshold of $t = 0.75$ which is small enough to recognize all true correspondences resulting in a high recall (see also Sect. 5). However, the lower the threshold the more correspondences are usually created which normally affects the precision negatively. To reduce the number of correspondences and to increase the precision we apply a filtering step in the post-matching process. The result of the filtering step (see Algorithm 4) is a 1:1 mapping containing a correspondence for each data item to the most similar counterpart which is not used by another correspondence. An equivalent filter algorithm is described in [Me04].

3.3 Schema Merging

Mapping the schema of a second input form to the same target table implies that the associated target schema needs to be adapted. Following our concept, there are two situations in which the target schema is adapted. First, schema elements of the new source schema for which no counterpart in the current target schema could be found are associated with new schema elements that are added to the target schema. The target schema increases since only table columns can be added; we avoid column deletions in the mapping generation process because they would result in loosing data. Other possible schema modifications, such as data type transformations, are reported to and solved in the user-based mapping validation, the second process in which the target schema can be adapted. In this process, the user can verify and refine the created mapping. Column deletions would result when the user maps a new column of the source schema for which the mapping process has not found a counterpart and, thus, a new target column has been created. Since the validation process is necessary before the target schema is implemented and data is loaded to the target schema, the deletion of schema elements is only an intermediate step. In case the mappings to a single target table needs to be completely reorganized, we firstly remove and recreate the target table and then load relevant data from source systems afterwards.

4 Schema Matching Infrastructure

We utilize a service-based infrastructure to manage source schemas including corresponding metadata of input forms. Fig. 4 shows an overview of the infrastructure consisting of different input systems, the MDR and the Research Database. There are two types of input systems differentiated by their functionality to use and provide metadata of input forms. We use LimeSurvey [Li16] as online input system which is currently the only system providing metadata and data. The commercial system Teleform is used to design paper-based input forms which can be automatically scanned and converted (OCR) after they have been filled with data by study participants. Although, Teleform system manages metadata about the designed input form, we were unable to use them since form designers in *LIFE* used the graphical mode to design forms which brings out no utilizable metadata. Therefore, we manually created metadata according to the designed input forms. There is a large amount of external sources including laboratory data and those generated by medical devices and/or provided in spreadsheets etc. The MDR imports metadata about input forms and schema directly from the API of the LimeSurvey system. The metadata of other input systems are specified using a spreadsheet template. All collected metadata data are accessible by a comprehensive REST-ful web-service interface. We use a web-based application to import metadata as well as creating and managing mappings. Mappings can be downloaded and distributed to research groups who have provided input forms of an assessment. They validate and correct mappings; mappings within the repository are updated when mapping spreadsheets are returned and uploaded. We will work on a web-application allowing to adapt mappings directly instead of using spreadsheets.

A fundamental part of the MDR infrastructure is the central repository. Fig. 5 shows the high-level relational schema of this repository. It consists of two main parts for managing schemas, code lists and their mappings. For the first part, the schema and mapping portion,

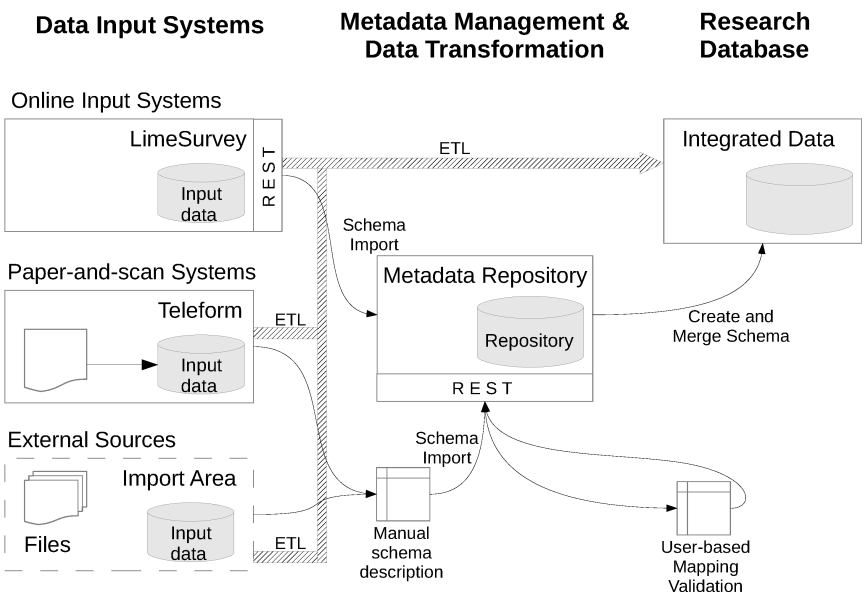


Fig. 4: Data Flows of scientific Data and Metadata - Use of the MDR

we explicitly differentiate between source and target schemas (see repository tables holding source and target table and column metadata). Unlike target schema, source schemas contain selected descriptions of the corresponding input form, i.e., information about the form, groups and items. This schema portion is denormalized but simplifies the import and querying and will never updated since source schemas of input forms will never change. Each imported source schema and created target schema is associated with a source (table *Source*). We additionally describe sources by connection information allowing to access sources, e.g., when the target schema is manipulated and when the data are extracted, transformed and loaded from sources to the target.

Code lists and transformation expressions are managed in the second portion of the repository schema. We differentiate between code lists of sources and target. Source code lists are defined when the input forms and their corresponding schema is imported whereas target code lists are created and adopted by the match process. Target code lists can be adapted by users when the corresponding schema mapping is validated to modify codes and labels, e.g., to replace alphanumeric with numeric codes. During the mapping generation process a mapping is created interrelating source and target code lists. The code list mapping is then used in the ETL process when data need to be transformed.

There are columns in source and target schemas which are not associated with code lists (e.g., text input such as comments or numeric values). To transform values of such columns we define and implement database functions on demand. For example, we implemented general functions converting values between different data types but also specific functions, such as transforming values from pounds to kilograms. Moreover, we also experienced the requirement to transform only specific values. Such values are typically expressions

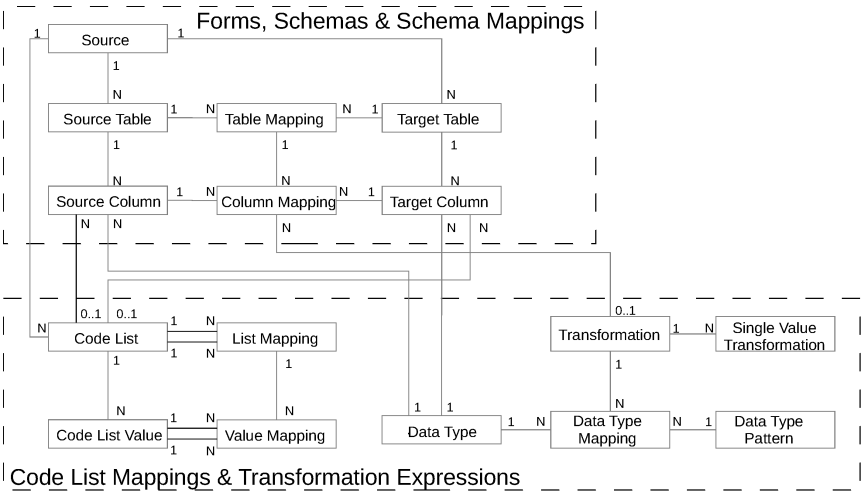


Fig. 5: High-level Database Schema of the Metadata Repository

for a special state, e.g., to express that a value could not be measured etc. Such "default" values can be differently defined by several groups designing the input forms but should be harmonized when the data are transferred to the target. Therefore, we pre-define value mappings (table *Single Value Transformation*) defining which source value is transformed into which target value. Transformations are associated to column mappings and, thus, are not globally executed per default.

Each source and target column is associated with the data type it is implemented. However, different database management systems including MySQL, MSSQL, and Oracle provide similar data types which differ at least in their name (MySQL: VARCHAR → Oracle: VARCHAR2). While data types of a source schema column are associated (table *Data Type*) in the import process, we provide a predefined data type mapping to construct the target schema using the correct data type. Unlike other mappings, the data type mappings do not map two equivalent data types but associate data type patterns to concrete target data types. A data type pattern represents multiple data types using the same type (e.g., VARCHAR, NUMERIC) but varying in their length and precision; length and precision are restricted by an upper and lower bound. Using data type patterns instead of concrete data types necessitates only few predefined data type mappings to address type differences of several database management systems. All these kinds of transformations can be specified for a single correspondence (table *Column Mapping*) and influences the target schema creation or adoption as well as data transformation processes.

5 Evaluation

In this section, we first give an overview about metadata and then discuss selected results.

5.1 Evaluation Data and Setup

System and matcher evaluation is typically difficult in the absence of a gold standard of an at least similar scenario. We use metadata and available mappings for the evaluation. This data has been collected over the last five years (since 2011). Mappings are verified by a small set of persons who have been qualified by special internal trainings. Therefore, we use these available mappings as gold standard for the evaluation. Tab. 1 shows an overview of the assessment and input form quantity structure per input systems and examination type. Most assessments has been implemented by input forms in LimeSurvey. The Teleform system is mainly used for questionnaires. Laboratory Data and input forms for process documentation is captured by our in-house LIMS (Laboratory Information Management System). All other data sources, i.e., spreadsheets, desktop databases etc. are subsumed by "External Sources". Especially in LimeSurvey and Teleform, there are 2 to 4 input forms per assessment. Assessments of the LIMS and of external sources are mainly implemented by a single input form. These forms are typically "virtual" forms, i.e., they are created to capture data generated by laboratory devices (within the wet lab) or are extracted, prepared, and imported from medical devices into an import schema. An exception includes physical examination forms (no laboratory results) of the LIMS. These forms are input forms to document specimen extraction processes. The complexity of assessments (measured by average number of items per form) differs widely. It ranges from 10 items (input forms for laboratory processes) to approx. 170 items used in interviews and, thus, does not depend on the input system but on the examination type. We also see that complex assessments are mainly implemented in multiple input forms. Hence, a manual mapping generation is a very resource-intensive task.

Data Source	**Examination Type**	**|Assessments|**	**|Forms|**	**avg (|Forms per Assessment|)**	**avg (|Items per Form|), (min - max)**
LimeSurvey	Interview	88	195	2.2	133.43 (19 - 844)
	Questionnaire	287	568	1.9	53.44 (13 - 728)
	Phys. Examination	218	438	2.0	74.20 (13 - 789)
Teleform	Questionnaire	59	194	3.3	86.51 (37 - 411)
	Phys. Examination	1	4	4.0	36.75 (36 - 37)
LIMS	Laboratory Data	121	123	1.0	20.56 (14 - 191)
	Phys. Examination	57	89	1.6	10.63 (8 - 17)
External Sources	Interview	3	3	1.0	105.00 (18 - 206)
	Laboratory Data	1	1	1.0	25.00 (25 - 25)
	Questionnaire	18	18	1.0	39.17 (16 - 153)
	Phys. Examination	71	74	1.1	48.35 (6 - 276)

Tab. 1: Quatity Structure of Input Forms and Items per Examination Type and Input System

|Mappings|	|Assessments|	avg (|Items per Assessment|)	min - max(|Items per Assessment|)
1	528	47.5	6 - 789
2	138	78.9	6 - 844
3	62	59.7	9 - 279
4	37	101.9	19 - 637
5	31	73.6	16 - 463
6	22	74.6	19 - 411
7	22	58.9	17 - 425
8	12	83.6	20 - 467
9	2	65.6	27 - 197
10	1	102.9	83 - 119

Tab. 2: Mapping Quantity

The idea of Tab. 2 is to show the number and complexity of generated mappings. In particular, it counts assessments (target schemas) in column 2 which are associated with a given number of input forms (= |source schema| = |mappings|) in column 1. For example, there are 538 assessments interrelated with only a single input form, 138 assessments which are associated with two input forms and so on. Hence, more than the half of available assessments are associated with a single input form for which the match generation process has not been applied. This set includes old input forms which are used as a test form but also new input forms which haven't changed so far. The number of assessments decreases for an increasing number of mappings. There is only one assessment available which is associated to 10 input forms, each contains on average 103 items.

5.2 Quality Evaluation

The goal of this evaluation is to analyze the quality of the match generation process. The evaluation utilizes the available and manually checked mappings from the productive instance as gold standard; we only include assessments to which two or more input forms are associated (see Tab. 2 column 1). We created mappings by varying the distance measure and the threshold. We implemented different trigram distance measures and the Levenshtein distance; distances are converted into similarities $0 \leq sim(x, y) \leq 1 \subset R$. The similarity value 0 means that x and y are maximal distant and the value 1 stands for equality of x and y. Fig. 6 consists of three charts showing precision, recall and F-measure for the for different distance measures and threshold values ranging from 0.5 to 1.0 (stepwidth 0.1). All utilized distance measures have a higher precision than 95%; $Trigram_{Jaccard}$ shows the best precision over all measured thresholds while Levenshtein distance measure result in smallest precision values but with a small offset to $Trigram_{Jaccard}$. In constrast to precision, Levenshtein's recall values are on the high end while those of $Trigram_{Jaccard}$ are on the lower end. The lowest recall (63%) obtain all distance measures for the threshold 1.0. Except $Trigram_{Jaccard}$ all used distance measures achieve an F-measure higher than 92% using a threshold from 0.5 to 0.9. The lowest F-measure (77%) for all distance measures is obtained for threshold 1.0. This means that there are only minimal changes in question texts

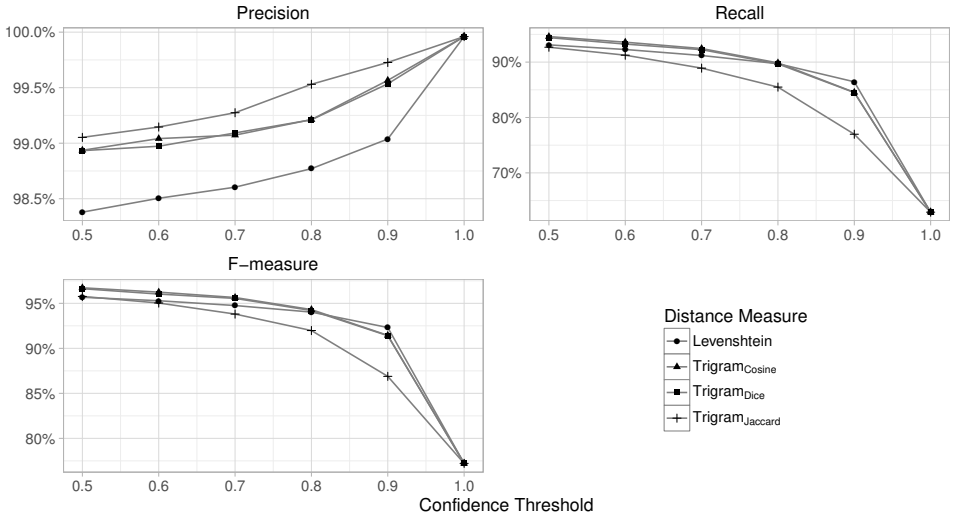


Fig. 6: Quality Evaluation Results using Precision, Recall and F-Measure

of corresponding data items between different input forms. Larger text changes normally result in a shifted meaning of data item and, thus, are also mapped to a new target column. However, diving into the results shows that our approach doesn't recognize negations, i.e., data items using nearly the same question text and same code lists but the meaning is turned around. The word "not" is too short to influence the distance measures. Moreover, there is also a small set of data items which utilize images instead of question text. The MDR doesn't import images, i.e., the item description (question text) for such data items is empty. Hence, mappings between such items are typically false positive by chance.

5.3 Blocking Evaluation

The goal of this evaluation is to analyze the efficiency of the blocking strategy used by our approach. There are different other potential blocking strategies; Fig. 7 shows the number of comparisons (during mapping generation process) using these blocking strategies. Note, a matching is only executed when there are multiple input forms per assessment available (see Tab. 2). A brute force strategy doesn't take a blocking into account and, thus, compares each items of each assessment with each other item (independently from the assessment). The resulting high number ($6,982.7 * 10^6$) of item comparisons can be decreased by a factor of 146 (to $47.8 * 10^6$ comparisons) when generating the cross product of items of input forms belonging to the same assessment (target). We call this strategy *assessment-wide brute force*. Next, this number of comparisons can be halved (to $23.9 * 10^6$ comparisons) when a matching is only executed between items of unmatched and already matched input forms. Computing correspondence similarities of items of a single unmapped input form with items of a single but already matched input form, e.g., the previous form as described above, results in, again, the half number of correspondences ($11.4 * 10^6$ comparisons).

This number of comparisons can be continually decreased by a factor of 3 (to $3.8 * 10^6$) when only items of equally named item groups between different input forms are matched. This is an overall reduction factor of 1,838 to the brute force strategy. This group blocking strategy is very efficient but simultaneously restrictive which could result in missing correspondences when two item groups are differently named but containing similar items for which a mapping should be generated. However, as decribed in Sect. 3, subsequent input forms of a single input system are usually copied and modified such that group names typically keep unchanged. Only group names of input forms between different input systems need to be synchronized.

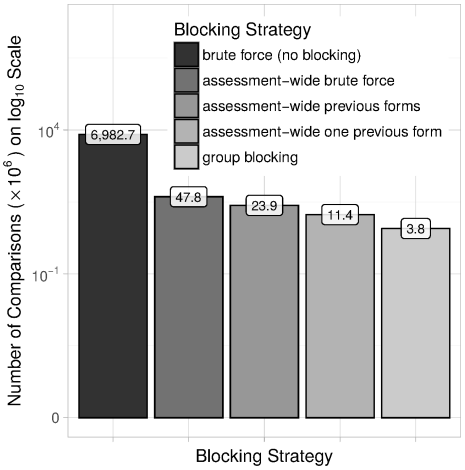


Fig. 7: Evaluation of the Blocking Efficiency

6 Related Work

There have been several project grants over the last one and a half decades with the goal to scientifically determine metadata management in medical sciences and to design and implement tools managing metadata (see e.g. [St09]). The National Metadata Repository (N-MDR) is an attempt to create a software toolset to be used in medical research. It utilizes metadata foundations described by the ISO standard 11179 [IS16]. However, there were no implementations of the N-MDR available when *LIFE* has been started 2009. The same holds for the MDM-Portal [Du16] containing a large amount of input forms. The Operational Data Model (ODM) [Ku09] is a data exchange format provided by the CDISC consortia. All, the N-MDR, ISO standard and the ODM format, describe metadata on assessment level but don't contain metadata structures for input forms and their mappings to target assessments.

A large amount of related work on schema matching has been published over the last one and a half decades. Selected surveys [RB01, Me04, Do06, ES07, DR07, LN07, BBR11] give a good introduction, overview and recent advances of the topic. Numerous software tools have been designed and implemented for schema matching including Cupid [MBR01], COMA/ COMA++ [DR02, Au05], and AgreementMaker [CAS09] to name only a few of them. However, many tools automatically compute schema mappings and neither capture explicit user feedback nor implement and adopt the target schema. Moreover, several tools can only be executed in stand-alone mode. Our MDR is implemented as a service-based infrastructure; it runs as service and interacts with other applications in our production environment over a REST-ful web-service interface.

Like model-management approaches [Be03, Me05, QKL07] and tools, such as RONDO [MRB03], our approach derives and implements the target schema in a given relational database. In contrast to these approaches we only add new columns to the target schema

and recreate the target schema or portions of it when other operations (attribute changes or deletions) are required. This strategy is simple but does not require powerful model-management-operators as introduced in [Me04].

7 Conclusion

In this paper, we presented an approach to manage metadata for data integration in medical sciences. The data are captured by different input forms and systems. Potentially, there are multiple input forms per assessment. They belong to different input systems but also structurally change over time. Metadata about input forms and schemas are collected and centrally managed by a Metadata Repository (MDR). Based on these metadata, the MDR automatically generates target schemas and schema mappings in a two step process. The first step derives the target schema from a schema of a selected input form. The second step iteratively creates mappings between schemas of input forms and the target schema by mapping data items of input forms. Target schema and schema mappings are extensively used by the ETL process to transfer data from data source to the research database. Moreover, the metadata are used to automatically create quality reports and to generate annotated schema forms.

Our approach is completely implemented using a service-based infrastructure running in a production environment of the large bio-medical research project *LIFE*. It has been applied for many input forms resulting in a target database with more than 900 assessments (target tables). We evaluated the schema matching approach by using manually verified mappings. In the future, we will associate data items of the target schema with public ontologies and terminologies as prerequisite to exchange and compare data with other studies on item level.

Acknowledgment

This publication is supported by LIFE - Leipzig Research Center for Civilization Diseases, Universität Leipzig. LIFE is funded by means of the European Union, by the European Regional Development Fund (ERDF) and by means of the Free State of Saxony within the framework of the excellence initiative.

References

[Au05] Aumüller, David; Do, Hong-Hai; Massmann, Sabine; Rahm, Erhard: Schema and ontology matching with COMA++. In: Proc. of the SIGMOD Conference. 2005.

[BBR11] Bellahsene, Zohra; Bonifati, Angela; Rahm, Erhard: Schema Matching and Mapping. Springer Verlag, 2011.

[Be03] Bernstein, Phil A.: Applying Model Management to Classical Meta Data Problems. In: Proc. of the CIDR. 2003.

[CAS09] Cruz, I.; Antonelli, F.; Stroe, C.: AgreementMaker: Efficient Matching for Large Real-World Schemas and Ontologies. In: Proc. of the VLDB. 2009.

[Do06] Do, Hong-Hai: Schema Matching and Mapping-based Data Integration. Verlag Dr. Müller (VDM), 2006.

[DR02] Do, Hong-Hai; Rahm, Erhard: COMA - A System for Flexible Combination of Schema Matching Approaches. In: Proc. of the VLDB. 2002.

[DR07] Do, Hong-Hai; Rahm, Erhard: Matching large schemas: Approaches and evaluation. Information Systems, 2007.

[Du16] Dugas, Martin; Neuhaus, Philipp; Meidt, Alexandra; Doods, Justin; Storck, Michael; Bruland, Philipp; Varghese, Julian: Portal of Medical Data Models: Information Infrastructure for Medical Research and Healthcare. Database, 2016, 2016.

[ES07] Euzenat, Jérôme; Shvaiko, Pavel: Ontology Matching. Springer Verlag, 2007.

[IS16] ISO 11179. http://metadata-standards.org/11179, Last online access, October, 02 2016.

[Ku09] Kuchinke, W.; Aerts, J.; Semler, S. C.; Ohmann, C.: CDISC Standard-based Electronic Archiving of Clinical Trials. Methods of Information in Medicine, 48(5):408 – 413, 2009.

[Li16] LimeSurvey - The Open Source Survey Application. http://www.limesurvey.org, Last online access, October, 02 2016.

[LN07] Legler, F.; Naumann, Felix: A Classification of Schema Mappings and Analysis of Mapping Tools. In: Proc. of the 12th BTW Conference. 2007.

[Lo15] Loeffler, Markus; Engel, Christoph; Ahnert, Peter et al.: The LIFE-Adult-Study: Objectives and Design of a population-based Cohort Study with 10,000 deeply Phenotyped Adults in Germany. BMC Public Health, 15, 2015.

[MBR01] Madhavan, Jayant; Bernstein, Phil A.; Rahm, Erhard: Generic Schema Matching With Cupid. In: Proc. of the VLDB. 2001.

[Me04] Melnik, Sergey: Generic Model Management: Concepts and Algorithms, volume 2967 of LNCS. Springer Verlag, 2004.

[Me05] Melnik, Sergey; Bernstein, Phil A.; Halevy, Alon; Rahm, Erhard: Supporting Executable Mappings in Model Management. In: Proc. of the SIGMOD Conference. 2005.

[MRB03] Melnik, Sergey; Rahm, Erhard; Bernstein, Phil A.: Rondo: A Programming Platform for Generic Model Management. In: Proc. of the SIGMOD Conference. 2003.

[QKL07] Quix, Christoph; Kensche, D.; Li, X.: Generic Schema Merging. In: Proc. of the CAISE. 2007.

[Qu12] Quante, Mirja; Hesse, Maria; Döhnert, Mirko; Fuchs, Michael; Hirsch, Christian; Sergeyev, Elena; Casprzig, Nora; Geserick, Mandy; Naumann, Stephanie; Koch, Christiane; Sabin, MA; Hiemisch, Andreas; Körner, Antke; Kiess, Wieland: The LIFE Child Study: a Life Course Approach to Disease and Health. BMC Public Health, 12(1):1021, 2012.

[RB01] Rahm, Erhard; Bernstein, Phil A.: A Survey of Approaches to Automatic Schema Matching. VLDB Journal, 2001.

[St09] Stausberg, Jürgen; Löbe, Matthias; Verplancke, Philippe; Drepper, Johannes; Herre, Heinrich; Löffler, Markus: Foundations of a Metadata Repository for Databases of Registers and Trials. In: MIE. pp. 409–413, 2009.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 195

A Hybrid Approach for Efficient Unique Column Combination Discovery

Thorsten Papenbrock[1] Felix Naumann[2]

Abstract: Unique column combinations (UCCs) are groups of attributes in relational datasets that contain no value-entry more than once. Hence, they indicate keys and serve data management tasks, such as schema normalization, data integration, and data cleansing. Because the unique column combinations of a particular dataset are usually unknown, UCC discovery algorithms have been proposed to find them. All previous such discovery algorithms are, however, inapplicable to datasets of typical real-world size, e.g., datasets with more than 50 attributes and a million records.

We present the hybrid discovery algorithm HyUCC, which uses the same discovery techniques as the recently proposed functional dependency discovery algorithm HyFD: A hybrid combination of fast approximation techniques and efficient validation techniques. With it, the algorithm discovers *all minimal unique column combinations* in a given dataset. HyUCC does not only outperform all existing approaches, it also scales to much larger datasets.

Keywords: unique column combinations, data profiling, metadata, hybrid.

1 Unique Column Combinations

A unique column combination (UCC) is a set of attributes whose projection contains no duplicate entry. Knowing these unique combinations is particularly important when choosing key constraints for a given relational dataset, because the values in such columns uniquely identify all records in the dataset. Unique column combinations are moreover known to be useful for schema normalization, data cleansing, query optimization, schema reverse engineering, and many other tasks.

If not explicitly declared as key constraints, the UCCs of a particular dataset are typically unknown and need to be discovered. This is particularly true in "data lake" scenarios, which involve many external data sources. Data profiling is the computer science discipline that describes the investigation of a dataset for its metadata [AGN15]. The discovery of UCCs is an important profiling task, which has led to the development of various discovery algorithms [He13, Si06, AN11]. The task for these algorithms is to find *all minimal UCCs* that hold in a given relational instance. The search is restricted to minimal UCCs, i.e., sets of attributes from which no attribute can be removed without invalidating the uniqueness of the described column combination, because all non-minimal UCCs can easily be derived from the set of minimal UCCs. Furthermore, most use cases, such as database key discovery [Ma16], are interested in only the minimal UCCs.

[1] Hasso Plattner Institute (HPI), Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, thorsten.papenbrock@hpi.de
[2] Hasso Plattner Institute (HPI), Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, felix.naumann@hpi.de

The discovery of UCCs is a computational expensive task, because the search is NP-hard [Gu03] and even the solution space is exponential [He13]. For this reason, all known algorithms are limited to small datasets. We propose a new UCC discovery algorithm called HyUCC that takes UCC profiling a step forward, now being able to efficiently process datasets that are much larger than current limits. In fact, with HyUCC the discovery time becomes less of an issue than the ability of the executing machine to cope with the size of the UCC result set, which can grow exponentially large.

In a recent publication [PN16], we proposed the HyFD algorithm for the discovery of functional dependencies. Its fundamental ideas are also the basis for our new algorithm HyUCC, because FD discovery and UCC discovery are quite similar, i.e., many FD discovery techniques can also be used to profile UCCs and vice versa. So the main contributions of this short paper can be summarized as follows:

1. *Hybrid UCC algorithm*. We present a hybrid algorithm for the discovery of all minimal unique column combinations in relational datasets: The algorithm combines known row- and column-efficient techniques to cope with both long and wide datasets.

2. *Evaluation*. We evaluate our algorithm on several datasets demonstrating its superiority over existing UCC discovery algorithms. The experiments show that the algorithm is capable of computing both large numbers of rows and columns.

We first discuss related work in Section 2. Then, Section 3 introduces the intuition of our hybrid approach. Section 4 describes how these ideas can be implemented by explaining the differences to HyFD. In Section 5, we evaluate our algorithm and conclude in Section 6.

2 Related Work

There are basically two classes of UCC discovery algorithms: row-based discovery algorithms, such as GORDIAN [Si06], and column-based algorithms, such as HCA [AN11]. Row-based algorithms compare pairs of records in the dataset, derive so-called agree or disagree sets, and finally derive the UCCs from these sets. This discovery strategy performs well with increasing numbers of attributes, but falls short when the number of rows is high. Column-based algorithms model the search space as a powerset lattice and then traverse this lattice to identify the UCCs. The traversal strategies usually differ, but all algorithms of this kind make extensive use of pruning rules, i.e., they remove subsets of falsified candidates from the search space (these must be false as well) and supersets of validated candidates (which must be valid and not minimal). The column-based family of discovery algorithms scales well with larger numbers of records, but large numbers of attributes render them infeasible. Because both row- and column-efficient algorithms have their strengths, we combine these two search strategies in our HyUCC algorithm.

The currently most efficient UCC discovery algorithm is DUCC [He13], a column-based algorithm. The algorithm finds UCCs with a random walk approach through the search space lattice making maximum use of the known pruning rules. Because this algorithm has shown to be faster than both GORDIAN and HCA, it serves as our evaluation baseline.

The HyFD algorithm is a hybrid FD discovery algorithm that already proposed to mix row- and column efficient discovery techniques in order to scale with both dimensions of a relational dataset [PN16]. In a sense, the HyUCC algorithm is the sister algorithm of HyFD, which we modified in certain selected components to find UCCs instead of FDs. The changes we made are presented in this paper; the achieved performance improvements are comparable, as our evaluation shows.

3 Hybrid UCC discovery

The core idea of hybrid UCC discovery is to combine techniques from column-based and row-based discovery algorithms into one algorithm that automatically switches back and forth between these techniques, depending on which technique currently performs better. The challenge for these switches is to decide when to switch and to convert the intermediate results from one model into the other model, which is necessary to let the strategies support each other. In the following, we first describe the two individual discovery strategies; then, we discuss when and how the intermediate results can be synchronized.

Row-efficient strategy. Column-based UCC discovery algorithms, which are the family of algorithms that perform well on many rows, model the search space as a powerset lattice of attribute combinations where each edge represents a UCC candidate. The search strategy is then a classification problem of labelling each node as non-UCC, minimal UCC, or non-minimal UCC. Figure 1 depicts an example lattice for five attributes A, B, C, D, and E with labeled nodes.

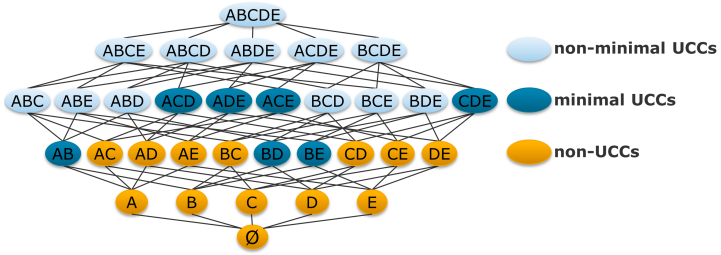


Fig. 1: UCC discovery in a powerset lattice.

For our hybrid algorithm, we propose a simple bottom-up traversal strategy: First, we test all candidates of size one, then of size two and so on. The lattice is generated level-wise using the *apriori-gen* algorithm [AS94]. Minimality pruning assures that no implicitly valid UCCs, i.e., non-minimal UCCs are ever generated [He13]. All discovered minimal UCCs must be stored as the algorithm's result.

An important characteristic of this discovery strategy is that all intermediate results during the discovery are correct but incomplete, that is, each discovered UCCs must be valid but not all UCCs have been discovered. Because correctness is guaranteed, we will always end the hybrid algorithm in a phase with this discovery strategy. Another characteristic of the bottom-up lattice traversal is that it might have to wade through many non-UCCs until

it reaches the true UCCs, because these are all placed along the virtual border between non-UCCs (below in the lattice) and true UCCs (above in the lattice). The fact that the number of these non-UCCs increases exponentially with the number of columns hinders algorithms of this family to scale well with increasing numbers of attributes – the lattice becomes extremely "wide". Hence, we need to utilize an alternative discovery strategy to skip most of the non-UCC nodes to reach the true UCCs faster.

Column-efficient strategy. Row-based / column-efficient UCC discovery strategies compare all records pair-wise and derive agree set from these comparisons. An agree set is a negative observation, i.e., a set of attributes that have same values in the two compared records and can, hence, not be a UCC; so agree sets correspond to non-UCCs in the attribute lattice. When all (or some) agree sets have been collected, there are efficient techniques to turn them into true UCCs [FS99].

A major weakness of this discovery strategy is that comparing all records pair-wise is usually infeasible. So suppose we stop the comparison of records at some time during the discovery; we basically compare only a sample r' of all r record pairs. When turning whatever agree sets we found so far into UCCs, these UCCs are most likely not all correct, because sampling might have missed some important agree sets. However, the intermediate result has three important properties (see [PN16] for proofs):

1. *Completeness:* Because all supersets of UCCs in the result are also assumed to be correct UCCs, the set of r'-UCCs is complete: It implies the entire set of r-UCCs, i.e., we find at least one X' in the r'-UCCs for each valid X in r-UCCs with $X' \subseteq X$.
2. *Minimality:* If a minimal r'-UCC is truly valid, then the UCC must also be minimal with respect to the real result. For this reason, the early stopping cannot lead to non-minimal or incomplete results.
3. *Proximity:* If a minimal r'-UCC is in fact invalid for the entire r, then the r'-UCC is still *close* to one or more valid specializations. In other words, most r'-UCCs need fewer specializations to reach the true r-UCCs on the virtual border than the unary UCCs at the bottom of the lattice so that the stopping approximates the real UCCs.

Hybrid strategy. For the hybrid UCC discovery strategy, we refer to the column-efficient search as the *sampling phase*, because we inspect carefully chosen subsets of record pairs for agree sets, and to the row-efficient search as the *validation phase*, because this phase directly validates individual UCC candidates. Intuitively, the hybrid discovery uses the sampling phase to jump over possibly many non-UCCs and the validation phase to produce a valid result. We obviously start with the sampling phase, switch back and forth between phases, and finally end with the validation phase. The questions that remain are *when* and *how* to switch between the phases.

The best moment to leave the sampling phase is when most of the non-UCCs have been identified and finding more non-UCCs becomes more expensive than simply directly checking their candidates. Of course, this moment is known neither a-priori nor during the process, because one would need to know the result already to calculate the moment. For

this reason, we switch optimistically back and forth whenever a phase becomes *inefficient*: The sampling becomes inefficient, when the number of newly discovered agree sets per comparison falls below a certain threshold; the validation becomes inefficient, when the number of valid UCCs per non-UCC falls below a certain threshold. With every switch, we relax this threshold a bit, so that the phases are considered efficient again. In this way, the hybrid discovery always progressively pursues the currently most efficient strategy.

To exchange intermediate results between phases, the hybrid algorithm must maintain all currently valid UCCs in some central data structure (we later propose a prefix-tree). When switching from sampling to validation, we update this central data structure of UCCs with the discovered agree sets. This means that we replace every single UCC for which a negative observation, i.e., an agree set exists with its valid, minimal refinements. The validation phase, then, directly operates on this data structure so that many non-UCCs are already excluded from the validation procedure. When switching from the validation to the sampling, the algorithm must not explicitly update the central data structure, because the validation already performs all changes directly to it. However, the validation automatically identifies record pairs that violated certain UCC candidates, and these record pairs should be suggested to the sampling phase for full comparisons as it is very likely that they indicate larger agree sets. In this way, both phases benefit from one another.

4 The HyUCC algorithm

We now describe our implementation of the hybrid UCC discovery strategy HyUCC. Because this algorithm is similar to the hybrid discovery algorithm HyFD, we omit certain details that can be found in [PN16]. The differences that make HyUCC discover unique column combinations instead of functional dependencies are in particular a prefix tree (trie) to store the UCCs, a UCC-specific validation, and UCC-specific pruning rules.

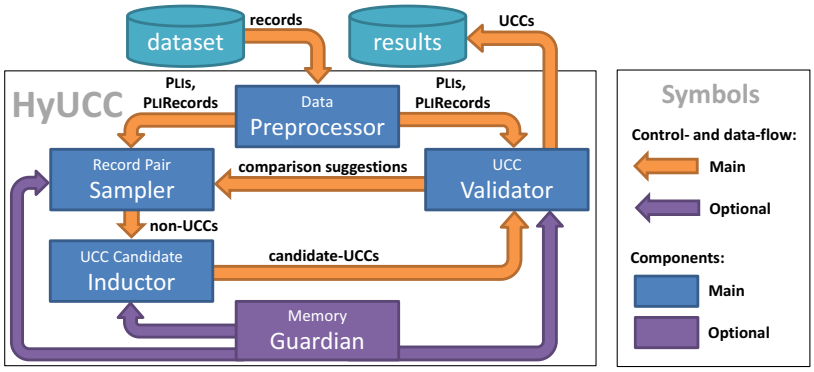


Fig. 2: Overview of HyUCC and its components.

Figure 2 gives an overview of HyUCC and its components. Given a relational dataset as a collection of records, the algorithm first runs them through a `Preprocessor` component that transforms the records into two smaller index structures: PLIs and PLIRecords. Then, HyUCC starts the sampling phase in the `Sampler` component. When the sampling has

become inefficient, the algorithms passes the discovered agree sets, i.e., the non-UCCs, to the `Inductor` component, which turns them into candidate-UCCs: UCCs that hold true on the sample of record pairs that was seen so far. Afterwards, the algorithm switches to the validation phase in the `Validator` component. This component systematically checks and creates candidate-UCCs. If the checking becomes inefficient, HyUCC switches back into the sampling phase handing over a set of comparison suggestions; otherwise, the validation continues until all candidates have been checked and all true UCCs can be returned. We now discuss the components of HyUCC in more detail.

Data Preprocessor. To determine a unique column combination, HyUCC must know the positions of same values in each attribute; it must not know the values itself. For this reason, the `Preprocessor` component transforms all records into the well-known and compact *position list indexes* (Plis) data structure (also known as stripped partitions [Hu99]). A Pli is a set of record ID sets, where each record in a set has the same value in the attribute described by the Pli. Using the Plis, the `Preprocessor` also creates PliRecords, which are the records from the input dataset with dictionary compressed values. These are needed for the record comparisons in the `Sampler` component.

Record Pair Sampler. The `Sampler` component compares the PliRecords to derive agree sets, i.e., non-UCCs. As stated earlier, a non-UCC is simply a set of attributes that have same values in two records. Because the sampling phase should be maximally efficient, the `Sampler` chooses the record pairs for the comparisons deliberately: Record pairs that are more likely to reveal non-UCCs are progressively chosen earlier in the process and less promising record pairs later. Intuitively, the more values two records share, the higher their probability of delivering a new non-UCC is. Vice versa, records that do not share any values cannot deliver any non-UCC and should not be compared at all.

So because the Plis already group records with at least one identical value, HyUCC only compares records within same Pli clusters. For all records within same Pli clusters, we must however define a possibly efficient comparison order. For this purpose, the `Sampler` component first sorts all clusters in all Plis with a different sorting key (see [PN16] for details). This produces different neighborhoods for each record in each of the record's clusters, even if the record co-occurs with same other records in its clusters. After sorting, the `Sampler` iterates all Plis and compares each record to its direct neighbor. In this step, the algorithm also calculates the *number of discovered non-UCCs per comparison* for each Pli. This number indicates the sampling efficiency achieved with this particular Pli. In a third step, the `Sampler` can then rank the different Plis by their efficiency, pick the most efficient Pli and use it to compare each record to its second neighbor. This comparison run updates the efficiency of the used Pli, so that it is re-ranked with the other. The `Sampler` then again picks the most efficient Pli for the next round of comparisons. This process of comparing records to their $n + 1$ next neighbors progressively chooses most promising comparisons; it continues until the top ranked Pli is not efficient any more, which is the condition to switch to the validation phase.

UCC Candidate Inductor. The `Inductor` component updates the intermediate result of UCCs with the non-UCCs from the `Sampler` component. We store these UCCs in a prefix

tree, i.e., trie, where each node represents exactly one attribute and each path a UCC. Such a *UCC tree* allows for fast subset-lookups, which is the most frequent operation on the intermediate results of UCCs. The UCC tree in HyUCC is much leaner than the FD tree used in HyFD, because no additional right-hand-sides must be stored in the nodes; the paths alone suffice to identify the UCCs.

Initially, the UCC tree contains all individual attributes, assuming that each of them is unique. The `Inductor` then refines this initial UCC tree with every non-UCC that it receives from the `Sampler`: For every non-UCC, remove the UCC and all of its subsets from the UCC tree, because these must all be non-unique. Then, create all possible specializations of each removed non-UCC by adding one additional attribute; these could still be true UCCs. For each specialization, check the UCC tree for existing subsets (generalizations) or supersets (specialization). If a generalization exists, the created UCC is not minimal; if a specialization exists, it is invalid. In both cases, we ignore the generated UCC; otherwise, we add it to the UCC tree as a new candidate.

UCC Validator. The `Validator` component traverses the UCC tree level-wise from bottom (individual attributes) to top (union of all attributes). This traversal is implemented as a simple breadth-first search. Each leaf-node represents a UCC candidate X that the algorithm validates. If the validation returns a positive result, the `Validator` keeps the UCC in the lattice; otherwise, it removes the non-UCC X and adds all XA to the tree with $A \notin X$ and XA is both minimal (XA has no specialization in the UCC tree) and valid (XA has no generalization in the UCC tree). After validating a level of UCC candidates, the `Validator` calculates the number of valid UCCs per validation. If this efficiency value does not meet a current threshold, HyUCC switches back into the sampling phase; the `Validator`, then, continues with the next level when it gets the control flow back.

To validate a column combination X, the `Validator` intersects the Plis of all columns in X. Intersecting a Pli with one or more other Plis means to intersect all the record clusters that they contain [Hu99]. The result is again a Pli. If this Pli contains no clusters of size greater than one, its column combination X is unique; otherwise, X is non-unique and the records in the clusters greater than one violate it. The algorithm suggests these records to the `Sampler` as interesting comparison candidates, because they have not yet been compared and may reveal additional non-UCCs of greater size.

An efficient way to calculate the Pli intersections is the following: First, pre-calculate the inverse of each Pli (this is done in the `Preprocessor` already). Then, take the Pli with the fewest records as a *pivot Pli* (recall that Plis do not contain clusters of size one so that the numbers of records usually differ). This pivot Pli requires the least number of intersection look-ups to become unique. For each cluster in the pivot Pli do the following: Iterate all record IDs and, for each record ID, look-up the cluster numbers in the inverted Plis of all other attributes in X; store each retrieved list of cluster numbers in a set. If this set already contains a cluster number sequence equal to the sequence the `Validator` wants to insert, then the algorithm found a violation and can stop the validation process for this candidate. In this case, the current record and the record referring to the cluster number sequence in the set are sent to the `Sampler` as a new comparison suggestion.

Memory Guardian. The `Guardian` is an optional component that monitors the memory consumption of HyUCC. If at any point the memory threatens to become exhausted, this component gradually reduces the maximum size of UCCs in the result until sufficient memory becomes available. Of course, the result is then not complete any more, but correctness and minimality of all reported UCCs is still guaranteed. Also, the result limitation only happens if the *result* becomes so large that the executing machine cannot store it any more. Other algorithms would simply break in such cases. To reduce the size, the `Guardian` deletes all agree sets and UCCs that exceed a certain maximum size. It then forbids further insertions of any new elements of this or greater size.

5 Evaluation

We evaluate and compare HyUCC to its sister algorithm HyFD [PN16] and to the UCC discovery algorithm DUCC, which the authors have shown to be superior over other approaches [He13]. All three algorithms have been implemented for the *Metanome* data profiling framework, which defines standard interfaces for profiling algorithms [Pa15]. The algorithms and the framework are available online[3]. Our experiments use a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB RAM. The server runs on CentOS 6.4 and uses OpenJDK 64-Bit Server VM 1.7.0_25 as Java environment. Details about our experimental datasets can be found in [PN16] and on our repeatability website[4]. Note that the experiments use the `null` = `null` semantics, because this was also used in related work; HyUCC can compute UCCs with `null` ≠ `null` as well, which makes the search faster, because columns with `null` values become unique more quickly.

5.1 Varying the datasets

In this experiment, we measure the discovery times for the three algorithms on eight real-world datasets. The datasets, their characteristics, and the runtimes are listed in Table 1. The results show that HyUCC usually outperforms the current state-of-the-art algorithm DUCC by orders of magnitude: On most datasets, HyUCC is about 4 times faster than DUCC; on the *flight* dataset, it is even more than 1,000 times faster. Only on *zbc00dt* DUCC is slightly faster, because only one UCC was to be discovered and DUCC does not pre-compute PliRecords or inverted Plis as HyUCC does. Furthermore, the runtimes of HyFD show that UCC discovery is considerably faster than FD discovery. Because HyFD and HyUCC use similar discovery techniques, this speedup is due to the smaller result sets.

Because we can easily parallelize the validations in HyUCC and HyFD, the experiment also lists the runtimes for their parallel versions. With 32 cores available, the parallel algorithms use the same number of threads. On the given datasets, HyUCC could in this way achieve 1.2 (*uniprot*) to more than 9 (*isolet*) times faster runtimes than its single-threaded version (less than 32, because only the validations run in parallel).

[3] `www.metanome.de`
[4] `www.hpi.de/naumann/projects/repeatability.html`

Dataset	Cols [#]	Rows [#]	Size [MB]	FDs [#]	UCCs [#]	DUCC	HyFD	HyUCC	HyFD parallel	HyUCC parallel
ncvoter	19	8 m	1,263.5	822	96	706.1	1,009.6	220.1	239.8	157.9
hepatitis	20	155	0.1	8,250	348	0.6	0.4	0.1	0.4	0.1
horse	27	368	0.1	128,727	253	0.8	5.8	0.2	3.7	0.2
zbc00dt	35	3 m	783.0	211	1	57.7	191.1	58.2	69.4	58.2
ncvoter_c	71	100 k	55.7	208,470	1,980	170.3	2,561.6	51.3	533.4	14.9
ncvoter_s	71	7 m	4,167.6	>5 m	32,385	>8 h	>8 h	>8 h	>8 h	5,870.2
flight	109	1 k	0.6	982,631	26,652	4,212.5	54.1	3.7	19.5	1.5
uniprot	120	1 k	2.1	>10 m	1,973,734	>8 h	>1 h	92.5	>1 h	76.7
isolet	200	6 k	12.9	244 m	1,241,149	>8 h	1,653.7	410.9	482.3	45.1

Tab. 1: Runtimes in seconds for several real-world datasets

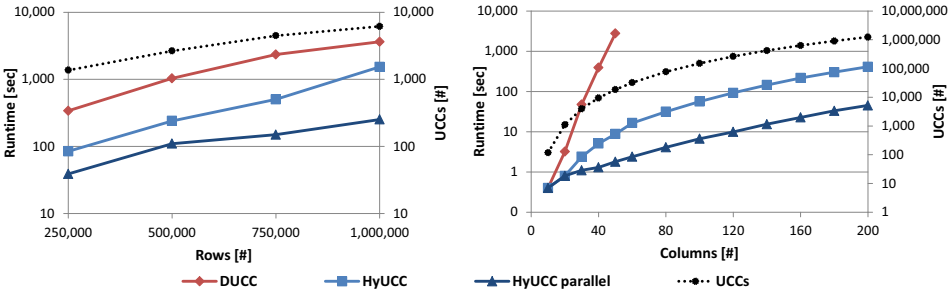


Fig. 3: Row scalability on *ncvoter_s* (71 columns) and column scalability on *isolet* (6238 rows).

5.2 Varying columns and rows

We now evaluate the scalability of HyUCC with the input's number of rows and columns. The row-scalability is evaluated on the *ncvoter_s* dataset with 71 columns and the column-scalability on the *isolet* dataset with 6238 rows. Figure 3 shows the measurements for DUCC and HyUCC. The measurements also include the runtimes for the parallel version of HyUCC; the dotted line indicates the number of UCCs for each measurement point.

The graphs show that the runtimes of both algorithms scale well with the number of UCCs in the result, which is a desirable discovery behavior. However, HyUCC still outperforms DUCC in both dimensions – even in the row-dimension that DUCC is optimized for: It is about 4 times faster in the row-scalability experiment and 4 to more than 310 times faster in the five column-scalability measurements that we could create for DUCC. HyUCC's advantage in the column-dimension is clearly the fact that the non-UCCs derived from the sampling phase allow the algorithm to skip most of the lower-level UCC candidates (and the number of these candidates increases exponentially with the number of columns); the advantage in the row-dimension is also this sampling phase of HyUCC, allowing it to skip many candidates and, because the number of UCCs also increases when increasing the number of rows, this gives HyUCC a significant advantage.

6 Conclusion & Future Work

In this paper, we proposed HyUCC, a hybrid UCC discovery algorithm that combines row- and column-efficient techniques to process relational datasets with both many records and many attributes. On most real-world datasets, HyUCC outperforms all existing UCC discovery algorithms by orders of magnitude.

For future work, we suggest to find novel techniques to deal with the often huge amount of results. Currently, HyUCC limits its results if these exceed main memory capacity, but one might consider using disk or flash memory in addition for these cases.

References

[AGN15] Abedjan, Ziawasch; Golab, Lukasz; Naumann, Felix: Profiling relational data: a survey. VLDB Journal, 24(4):557–581, 2015.

[AN11] Abedjan, Ziawasch; Naumann, Felix: Advancing the Discovery of Unique Column Combinations. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM). pp. 1565–1570, 2011.

[AS94] Agrawal, Rakesh; Srikant, Ramakrishnan: Fast Algorithms for Mining Association Rules in Large Databases. In: Proceedings of the International Conference on Very Large Databases (VLDB). pp. 487–499, 1994.

[FS99] Flach, Peter A; Savnik, Iztok: Database dependency discovery: a machine learning approach. AI Communications, 12(3):139–160, 1999.

[Gu03] Gunopulos, Dimitrios; Khardon, Roni; Mannila, Heikki; Saluja, Sanjeev; Toivonen, Hannu; Sharma, Ram Sewak: Discovering All Most Specific Sentences. ACM Transactions on Database Systems (TODS), pp. 140–174, 2003.

[He13] Heise, Arvid; Quiane-Ruiz, Jorge-Arnulfo; Abedjan, Ziawasch; Jentzsch, Anja; Naumann, Felix: Scalable Discovery of Unique Column Combinations. Proceedings of the VLDB Endowment, 7(4):301–312, 2013.

[Hu99] Huhtala, Ykä; Kärkkäinen, Juha; Porkka, Pasi; Toivonen, Hannu: TANE: An efficient algorithm for discovering functional and approximate dependencies. The Computer Journal, 42(2):100–111, 1999.

[Ma16] Mancas, Christian: Perspectives in Business Informatics Research. Springer, chapter Algorithms for Database Keys Discovery Assistance, pp. 322–338, 2016.

[Pa15] Papenbrock, Thorsten; Bergmann, Tanja; Finke, Moritz; Zwiener, Jakob; Naumann, Felix: Data Profiling with Metanome. Proceedings of the VLDB Endowment, 8(12):1860–1871, 2015.

[PN16] Papenbrock, Thorsten; Naumann, Felix: A Hybrid Approach to Functional Dependency Discovery. In: Proceedings of the International Conference on Management of Data (SIGMOD). 2016.

[Si06] Sismanis, Yannis; Brown, Paul; Haas, Peter J.; Reinwald, Berthold: GORDIAN: Efficient and Scalable Discovery of Composite Keys. In: Proceedings of the VLDB Endowment. pp. 691–702, 2006.

Data Analytics

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 207

Fast Approximate Discovery of Inclusion Dependencies

Sebastian Kruse[1] Thorsten Papenbrock[1] Christian Dullweber[2] Moritz Finke[2]
Manuel Hegner[2] Martin Zabel[2] Christian Zöllner[2] Felix Naumann[1]

Abstract: Inclusion dependencies (INDs) are relevant to several data management tasks, such as foreign key detection and data integration, and their discovery is a core concern of data profiling. However, *n*-ary IND discovery is computationally expensive, so that existing algorithms often perform poorly on complex datasets. To this end, we present Faida, the first approximate IND discovery algorithm. Faida combines probabilistic and exact data structures to approximate the INDs in relational datasets. In fact, Faida guarantees to find all INDs and only with a low probability false positives might occur due to the approximation. This little inaccuracy comes in favor of significantly increased performance, though. In our evaluation, we show that Faida scales to very large datasets and outperforms the state-of-the-art algorithm by a factor of up to six in terms of runtime without reporting any false positives. This shows that Faida strikes a good balance between efficiency and correctness.

Keywords: inclusion dependencies, data profiling, dependency, discovery, metadata, approximation

1 The Intricacies of Inclusion Dependency Discovery

It is a well-known fact that ever-increasing amounts of data are being collected. To put such large and complex datasets to use, be it for machine learning, data integration, or any other application, it is crucial to know the datasets' structure. Unfortunately, this information is oftentimes missing, incomplete, or outdated for all sorts of reasons. To overcome this quandary, the research area of *data profiling* has borne several algorithms to discover structural metadata of any given dataset.

A very important and fundamental type of structural metadata of relational databases are *inclusion dependencies (INDs)* [AGN15]. They form an integral component of foreign key (FK) discovery [Ro09], allow for query optimizations [Gr98], enable integrity checking [CTF88], and serve many further data management tasks. Intuitively, an IND describes that a combination of columns from one database table only contains values of another column combination, which might or might not be in the same table. Before looking at a concrete example, let us formalize this notion.

Definition 1 (Inclusion dependency) *Let r and s be two relational, potentially equal, tables with schemata $R = (R_1, \ldots, R_k)$ and $S = (S_1, \ldots, S_m)$, respectively. Further, let $\bar{R} = R_{i_1} \ldots R_{i_n}$ and $\bar{S} = S_{j_1} \ldots S_{j_n}$ be n-ary column combinations of distinct columns. We say that $\bar{R}$ is included in $\bar{S}$, i.e., $\bar{R} \subseteq \bar{S}$, if for every tuple $t_r \in r$, there is a tuple $t_s \in s$, such*

[1] Hasso Plattner Institute (HPI), Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, firstname.lastname@hpi.de
[2] Hasso Plattner Institute (HPI), Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, firstname.lastname@student.hpi.de

that $t_r[\bar{R}] = t_s[\bar{S}]$. $\bar{R}$ *is called the dependent column combination and* $\bar{S}$ *the referenced column combination. With both of them having n columns, the IND is said to be n-ary.*

Tab. 1 illustrates a lexicographical example dataset comprising a dictionary table that stores words of different languages, and a translation table that translates those words from one language to the other. Apparently, there are, amongst some others, two interesting, ternary INDs to be found in that example, namely *word1*, *lang1*, *type1* $\subseteq$ *word*, *lang*, *type* and *word2*, *lang2*, *type2* $\subseteq$ *word*, *lang*, *type*. Intuitively, these INDs require that all words in the translation table are found in the dictionary table. Note in particular the stronger semantics in comparison to INDs of lower arity, e.g., *word1* $\subseteq$ *word* and *word2* $\subseteq$ *word*. While the former, ternary INDs identify words not only by their literal but also their language and syntactical type, the latter, unary INDs merely consider the word literal, which does not suffice to uniquely identify a word (cf. *hat*). Due to these stronger semantics, it is worthwhile to discover INDs of the highest possible arity.

word	**lang**	**type**
hut	en	noun
hat	en	noun
has	en	verb
Hütte	de	noun
Hut	de	noun
hat	de	verb

(a) Dictionary table.

word1	**lang2**	**type1**	**word2**	**lang2**	**type2**	**fit**
hut	en	noun	Hütte	de	noun	$\perp$
hat	en	noun	Hut	de	noun	$\perp$
has	en	verb	hat	de	verb	$\perp$

(b) Translation table.

Tab. 1: A lexicographical example dataset with several INDs.

In the last years, several IND discovery algorithms have been proposed [Pa15, DMLP09, KR03, DMP03], pushing the boundaries in terms of efficiency and scalability. However, many real-world datasets cannot be processed by any of these algorithms within reasonable time, even on powerful hardware, for two main reasons: First, the number of valid *n*-ary INDs is often enormously large in real-world datasets. The result sets alone can, therefore, already exceed main memory limits [Pa15]. Second, and more commonly, the existing algorithms need to shuffle huge amounts of data to test IND candidates – in fact, the amount of shuffled data depends on the number of IND candidates and easily exceeds the inspected dataset in size. Some algorithms perform those shuffles out-of-core to overcome main memory limitations. Still, not only does this operation remain an efficiency bottleneck, but also the shuffled data can become so large that even disk storage limitations are exceeded!

We propose to tackle the latter issue by *approximating* the INDs of datasets, that is, for any given dataset we calculate a set of INDs that is complete but might contain false positives. However, the guarantee of correctness is traded for great performance improvements. Let us justify, why this trade is worthwhile: We observed that in real-world datasets any two column combinations are either related by an IND or their values are disjoint to a great extent. In other words, it is rare that the vast majority of values of one column combination are included in the other column combination, except for a small remainder. This clear cut allows to use more light-weight, approximate methods to test IND candidates without

risking severe accuracy losses. Nonetheless, in the few cases where two columns overlap in, say, 99 % of their values and an approximate method would indeed incorrectly report an IND, then this false positive is still a *partial IND*, i.e., it has only few violating values. We note that guarantueed and complete correctness of INDs is not required by many use cases, such as FK discovery [Ro09, Zh10] and data cleaning [Bo05].

To this end, we introduce FAIDA, the first approximate discovery algorithm for unary and n-ary INDs. FAIDA uses two different approximate data structures: a hash-based probabilistic data structure to characterize column combinations with a very small memory footprint, and a sampling-based inverted index to attenuate statistically expected inaccuracies. The combination of these two data structures offers high-precision results, because both compensate the other's weaknesses. In fact, we found FAIDA to report exact results in all our experiments. In addition, we characterize the novel class of *scrap INDs*, a sort of degenerate INDs that are not applicable to typical IND use cases but usually make up a considerable share of the INDs in a dataset. FAIDA identifies and prunes scrap INDs to narrow down the search space and achieve further performance improvements. This is particularly useful to prevail in situations where there are actually intractably many INDs, as mentioned above.

The remainder of the paper is organized as follows: In Sect. 2, we describe related work. We proceed to give an overview of FAIDA in Sect. 3, followed by a detailed description of its hybrid IND checking process in Sect. 4 and a formalization and rationale for scrap INDs in Sect. 5. Then, in Sect. 6, we compare FAIDA to the exact state-of-the-art IND discovery algorithm BINDER and evaluate FAIDA's effectiveness and efficiency in detail. Eventually, we conclude in Sect. 7.

2 Related Work

The discovery of dependencies, such as functional dependencies, order dependencies, or inclusion dependencies, in a given database is considered an essential component of data profiling [AGN15]. In this section, we focus on related work that addresses the approximate and exact discovery of inclusion dependencies.

Approximate IND discovery. We define approximation as estimation of the set of the actual INDs in a dataset. This nomenclature is in contrast to "approximate INDs" (also: "partial INDs") that hold only on a subset of the rows [LPT02, DMLP09]. This orthogonal concern is not the focus of this paper.

The only existing approach to approximate IND discovery is described by Zhang et al. as part of foreign-key (FK) discovery [Zh10]. It uses bottom-k sketches and the Jaccard index to approximate the inclusion of two columns based on Jaccard coefficients. Their approach has several disadvantages: For each level of n-ary INDs the hashes for the bottom-k sketches have to be computed from all actual values. Furthermore, it suffers from a similar problem as the probabilistic data structure used by FAIDA when comparing a column c_1 that has only few distinct values with a column c_2 that has many distinct values. The two bottom sets have potentially only a small overlap and in the worst case, all bottom hashes of c_2 are smaller than the bottom hashes of c_1. While FAIDA's errors are limited to false positives, this effect

can additionally lead to false negatives. Moreover, the authors focus on FK candidates and apply the approach only to relatively few candidates where the right-hand side has to be a known primary key. For these reasons, the proposed algorithm produces significantly different result sets than FAIDA, so that a performance comparison between these algorithms does not make sense.

Exact IND discovery. In previous research, much attention has been paid to the discovery of unary INDs, i.e., INDs between single columns. Different discovery strategies have been proposed, based on inverted indices [DMLP09], sort-merge joins [Ba07], and distributed data aggregation [KPN15]. While efficient unary IND discovery is an important part of n-ary IND detection, the problem is not of exponential complexity and therefore a much simpler task. Of course, FAIDA can also be applied to the efficient discovery of unary INDs.

Research has also devised exact algorithms for the discovery of n-ary INDs, in particular MIND [DMLP09] and BINDER [Pa15]. Both employ an Apriori-like discovery scheme to find IND candidates. While MIND tests these candidates individually against a database, BINDER employs a more efficient divide-and-conquer strategy to test complete candidate sets in a single pass over the data. This makes BINDER the current state-of the-art algorithm, which we compare against in our evaluation. Nevertheless, both strategies exhibit declined performance and increased memory consumption when testing IND candidates of high arity. In contrast, FAIDA, which also builds upon Apriori candidate generation, employs probabilistic data structures that do not suffer from this effect.

Besides Apriori-based approaches, depth-first algorithms have been proposed that optimize candidate generation towards inclusion dependencies of very high arity [KR03, DMP03]. However, these algorithms employ the same expensive IND checking mechanisms as MIND and are only applicable to pairs of tables, lacking a strategy to deal efficiently with whole datasets. Another recent approach avoids candidate generation entirely [SM16]. This is achieved by determining for every pair of tuples from two given tables, which unary INDs they support. These sets of unary INDs are then successively merged into maximal n-ary INDs. Again, no strategy is given to efficiently profile datasets with more than two tables.

Foreign key discovery. Although strongly connected, IND discovery and FK discovery are distinct problems: not every IND that holds on a given dataset is a FK relationship. Vice versa, in unclean databases, there might be semantically intended FK relationships whose corresponding INDs are violated by several tuples [Zh10]. However, INDs are a prerequisite for several FK discovery algorithms [Ro09, Zh10].

3 Overview of FAIDA

In this section, we present FAIDA, our Fast Approximate IND Discovery Algorithm, from a bird's eye view before giving more details in the following sections. Fig. 1 depicts FAIDA's general mode of operation. As for most n-ary IND discovery algorithms, FAIDA starts by identifying unary INDs, then uses an Apriori-style candidate generation process to generate binary IND candidates, and checks those in turn. This generate-and-test procedure

is repeated – i.e., the latest discovered, n-ary INDs are used to generate $(n+1)$-ary IND candidates, which are tested subsequently to retain the actual $(n+1)$-ary INDs – until the candidate generator produces no more IND candidates for some arity $n_{\max}$. In the following, we describe the various processing steps in more detail.

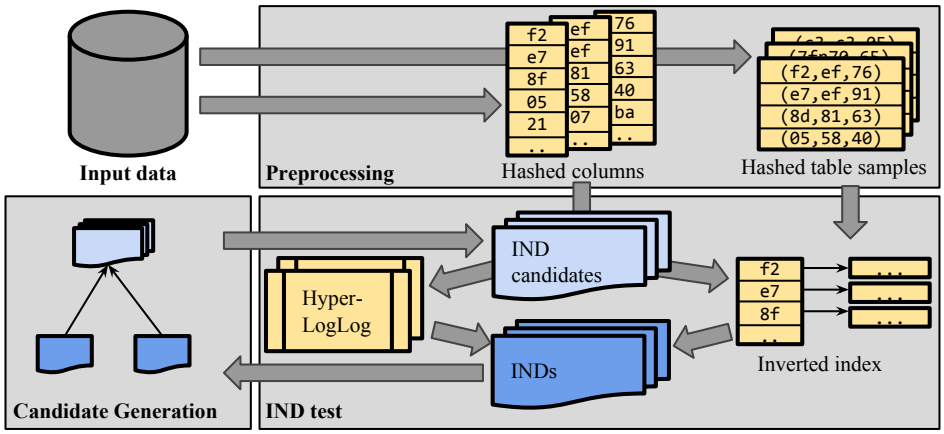


Fig. 1: Overview of FAIDA.

Preprocessing. The performance of IND discovery algorithms is mainly impacted by the fact that the input dataset has to be re-read and shuffled in every IND test phase. FAIDA attenuates this issue in a preprocessing step. At first, it converts the input dataset into *hashed columns*, i.e., the values in the input dataset tables are hashed and stored in a columnar layout. During the IND test phases, FAIDA will then resort to those hashed columns rather than the original input dataset, thereby greatly reducing the amount of data to be read, as we explain in the next paragraph. Furthermore, *hashed samples* of every table are stored – they are needed for bootstrapping the inverted index in the IND test phase. In Sect. 4.1, we explain the preprocessing step and its impact on performance and the IND result quality in greater detail. Still, we already want to remark that the use of compact hashes instead of actual values greatly improves performance of FAIDA in the subsequent phases, but it cannot guarantee exact results due to potential hash collisions. If two values share the same hash value, FAIDA will deem those two values equal. Nevertheless, this phenomenon can only cause false positive INDs but not miss out any INDs. Moreover, it is extremely unlikely that single hash collisions yield false positive INDs, because the distinction between INDs and non-INDs is usually not governed by single values only.

IND test. To test IND candidates, FAIDA uses a hybrid approach that builds upon a probabilistic *HyperLogLog* structure [Fl07] to represent columns with many distinct values and a sampling-based inverted index to represent columns with few distinct values. For each level, i.e., for each IND arity, FAIDA passes once over the relevant hashed columns, inserts them into the two data structures, and finally jointly evaluates them to determine the actual INDs. The IND tests might also produce false positives in addition to those caused by the hashing during preprocessing, but it does not produce false negatives. Because the IND

tests and the hashing have consistent error characteristics, FAIDA is guaranteed to find all INDs in a dataset. Sect. 4.2 to 4.3 discuss our hybrid IND test strategy in detail.

Candidate generation. FAIDA uses the same Apriori-style candidate generation as existing algorithms [DMLP09, Pa15]. This procedure makes use of the downward closure property of INDs: It only generates an n-ary IND candidate $A_1A_2 \ldots A_n \subseteq B_1B_2 \ldots B_n$, if the n $(n-1)$-ary INDs $A_2A_3 \ldots A_n \subseteq B_2B_3 \ldots B_n$, $A_1A_3 \ldots A_n \subseteq B_1B_3 \ldots B_n$, ..., and $A_1A_2 \ldots A_{n-1} \subseteq B_1B_2 \ldots B_{n-1}$ are verified to hold on the profiled dataset. There are multiple reasons why we did not replace the candidate generation with an approximate version. At first, we found in our experiments that the candidate generation only takes a tiny fraction of the overall runtime of IND algorithms - so the overall gains of any performance improvement here would be marginal. Secondly, if an approximate candidate generation produces false positives, we would likely end up with inferior performance, because the algorithm would need to test those additional IND candidates as well. Finally, if an approximate candidate generation yields false negatives, i.e., if it misses out on some IND candidates, FAIDA cannot guarantee completeness of its results anymore, which would be a bad trade. However, FAIDA might still prune some candidates deliberately: In Sect. 5, we describe the class of *scrap INDs* that oftentimes make up a great share of all INDs in a dataset but that are mostly useless. We show how to detect scrap INDs, so as to remove them from the set of IND candidates for performance improvements.

4 Fast and Lean Inclusion Dependency Approximation

As stated in Sect. 3, FAIDA adapts the same workflow for IND discovery as most exact algorithms: First, it discovers all unary INDs and, then, iteratively generates and tests IND candidates of respectively next arity. However, FAIDA uses approximation techniques in this process to reduce the amount of data handled in each iteration and, ultimately, to improve performance. In the following, we explain the building blocks as well as the interplay of this approximation scheme in more detail.

4.1 Read-Optimized Input Data

Whenever there is a set of IND candidates to be tested, exact IND discovery algorithms (i) read the input dataset, (ii) extract the value combinations that belong to dependent or referenced column combination of any IND candidate, and then (iii) shuffle those value combinations to compare the column combinations of the IND candidates to determine the actual INDs. By dropping the guarantee of the correctness of the discovered INDs, FAIDA can use a completely different, more efficient, and more scalable approach to test IND candidates. Some activities of FAIDA's IND test can be factored out of the IND test loop and instead be done only once, before the first IND test, which further improves performance. We describe those in the following.

Hashing. FAIDA's IND test uses hashes of the values in the input dataset, rather than the actual values. This is obviously favorable w.r.t. performance and scalability, because hashes

are of a small, fixed size in contrast to the actual values. Thus, they consume less memory and can be efficiently compared. Moreover, Faida's IND test uses HyperLogLog [Fl07] data structures, which operate on hashes anyway. However, depending on the hash function, the hashing can be quite CPU-intensive. Of course, it is necessary to read the input data once before it can be hashed. Because re-reading the input dataset over and over again is costly in terms of disk I/O, Faida reads the input dataset only once, hashes its values, and writes the resulting hashes back to disk. Note that for testing n-ary INDs with $n \geq 2$, other IND discovery algorithms need to shuffle *combinations* of values. In contrast, Faida merges the individual value hashes of those value combinations into a new, single hash value using a simple bitwise `XOR`. Consequently, the descriptions of Faida's other components refer, without loss of generality, only to single hash values and not to combinations of hash values.

Columnar data layout. In most cases, relational data is organized in a row layout. When testing n-ary IND candidates with $n \geq 2$, most columns of the input dataset usually do not appear in all IND candidates. Still, in a row layout, those columns have to be read without being of any use. Faida avoids this inefficiency by storing above described hashes in a columnar layout, thereby allowing the IND test to read only those columns that are part of an IND candidate. For the example dataset from Tab. 1, Faida creates nine files, each containing the hashes of the values of one of the columns in that dataset.

Table samples. As mentioned in Sect. 3, Faida uses a hybrid IND test strategy with HyperLogLog structures and an inverted index. The inverted index operates on a small sample of the (hashed) input data. Our algorithm calculates this sample once in the beginning and, then, reuses it in every IND test phase. In fact, we have the following requirements for the sample: Given a sample size s (e.g., $s = 500$), we need a sample of each table, such that this sample table contains $min\{s, d_A\}$ distinct values for each column A with d_A being the actual number of distinct values of A. The simple rationale for this requirement is that for columns with only few distinct values, we aim to ensure that these are effectively processed in the inverted index. If we took a random sample instead, we would most likely capture only a subset of its actual values leading to an impaired performance of the inverted index. To generate this sample, we use a simple greedy proceeding that is depicted in Algorithm 1.

The sampling algorithm is applied to each table individually and can be piggy-backed onto the above described preprocessing steps. Note that it operates on the hashed values and, thus, benefits from the low memory footprint of its data structures. The algorithm starts by initializing two data structures, namely T_s, which collects the sample tuples, and *sampledValues*, which tracks for each of the columns in the table the values that have been sampled from it so far (Lines 1–2). Then, it iterates all the tuples of the table (Line 3) to decide for each tuple if it should be included in the sample. A tuple should be included if a column exists that does not yet have s different sampled values and the tuple provides a yet unseen value for that column (Lines 4–7). If so, the tuple is added to T_s and the samples in *sampledValues* are updated accordingly. As an example, consider Tab. 1a and assume $s = 2$. In that case, Algorithm 1 would sample the first four tuples: The first tuple is always sampled anyway; the second tuple provides a new value for *word*; the third for *type*; and the fourth for *lang*. Afterwards, the algorithm has sampled at least two values for each column, so no further tuple will be picked.

Algorithm 1: Create a hashed table sample

Input : hashed tuples T for a table with attributes $A_1 \dots A_n$
minimum values s to be sampled per column
Output sample T_s of the hashed tuples
:

```
1  T_s ← ∅; sampledValues ← arrayOfSize(n);
2  foreach 1 ≤ i ≤ n do sampledValues[i] ← ∅ ;
3  foreach t ∈ T do
4      addToSample ← false;
5      foreach 1 ≤ i ≤ n do
6          if |sampledValues[i]| ≤ s ∧ t[A_i] ∉ sampledValues[i] then
7              addToSample ← true;
8      if addToSample then
9          T_s ← T_s ∪ {t};
10         foreach 1 ≤ i ≤ n do sampledValues[i] ← sampledValues ∪ {t[A_i]} ;
```

4.2 Scalable Probabilistic Inclusion Dependency Test

As stated in the previous sections, the major bottleneck of exact IND discovery algorithms is the IND candidate testing, which requires shuffling large amounts of data, especially when many IND candidates of higher arities arise. Not only are there more value combinations of larger individual size with increasing arity, but the shuffling itself also becomes more expensive. That is because the shuffling can eliminate duplicate values. However, the likelihood of duplicate value combinations drastically declines with the arity. Using hashes instead of values, as described in Sect. 4.1, can only mitigate but not completely avoid this problem and it might also eventually succumb to memory limitations.

Faida avoids the shuffling completely and uses a probabilistic approach for IND testing. The main idea is to calculate a summary for each column combination in the IND candidates and then perform a heuristic IND test on those summaries. An obvious instance of this idea is to encode column combinations with Bloom filters and then check for each IND candidate if its referenced column combination Bloom filter has all bits of the dependent column combination Bloom filter set. However, Bloom filters are prone to oversaturation for very large columns. Therefore, we use set cardinalities: Let $X \subseteq Y$ be an IND candidate and s a function that maps a column combination to the set of all its contained value combinations. Then $X \subseteq Y$ is an IND, if and only if $|s(Y)| = |s(X) \cup s(Y)|$. The set cardinality of a multiset can be efficiently and effectively estimated with *HyperLogLog* [Fl07], which scales to very large cardinalities with arbitrary precision.

Before we describe how Faida employs HyperLogLog for IND tests, let us briefly explain how that counting scheme works. At the core, it makes use of the following observation: Let s be a sample from a uniform distribution of the values from 0 to $2^k - 1$ for some k and let n be the number of leading zeroes in the binary representation of $\min s$ using k digits. Then $|s|$ can be estimated as 2^n. Assuming a good hash function that produces uniformly

distributed, mostly collision-free hash values for its input data, we can count any values via their hashes. HyperLogLog extends this idea by partitioning the hashes by their prefixes into buckets and maintain for each bucket the largest number of leading zeroes observed in the residual suffix bits. The observations in the different buckets are, then, merged via harmonic mean with additional bias correction [Fl07]. Consider Fig. 2 as an example, where we use HyperLogLog to estimate the set cardinality of the *word* column from Tab. 1. We employ a 4-bit hash function using the first bit for partitioning and the residual three bits to count leading zeroes. Apparently, for the bucket for the prefix `0`, `hash`(*has*) = `00`11 provides the most leading zeroes in the suffix (namely one), while for the bucket with the prefix `1`, `hash`(*hut*) = `1000` provides the most leading zeroes, namely three. Applying the harmonic mean and bias correction, HyperLogLog estimates four as the set cardinality of the input values. Note that HyperLogLog is, due to its stochastic nature, rather suited to estimate the set cardinality of larger datasets.

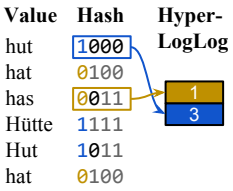


Fig. 2: Example HyperLogLog structure with two buckets.

Given this intuition of HyperLogLog, we proceed to show how we use this data structure for IND testing. As mentioned above, the basic idea is that for an IND candidate $X \subseteq Y$ to hold, the set cardinality of Y must be equal to the joint set cardinality of X and Y. A naïve implementation of this idea would use two HyperLogLog structures to estimate and compare both cardinalities, say HLL_Y and $\text{HLL}_{X \cup Y}$. This approach requires only very little processing effort in contrast to the shuffling of exact IND discovery algorithms: It is only necessary to scan once through the data and update the HyperLogLog structures, which themselves are of constant size. However, this is expected to work well only when (i) Y contains a stochastically relevant amount of elements and (ii) the differences of the count estimates of HLL_Y and $\text{HLL}_{X \cup Y}$ is greater than the expected estimation error of HLL_Y. The estimation error can be controlled via the number of buckets in the HyperLogLog structures. With our observation that the distinction of INDs and non-INDs is not governed by only few values, we can assume the second criterion to hold if we keep the expected estimation error small, e.g., around 0.1 %.

With that theoretical understanding on the applicability of HyperLogLog, we can now tailor it a bit more towards IND tests: The only case, where $\text{HLL}_{X \cup Y}$ would yield an estimate greater than that of HLL_Y, applies when there is some element in X that is not Y and that provides more leading zeroes to any partition in $\text{HLL}_{X \cup Y}$ than any element from Y does. Thus, we can instead maintain the two HyperLogLog structures HLL_X and HLL_Y and check if HLL_X has observed more leading zeroes than HLL_Y in any of the buckets. While this test is logically equivalent to the above naïve approach, it requires Faida to maintain only one HyperLogLog structure per column combination rather than up to two

HyperLogLog structures per IND, which is theoretically bounded only by the square of column combinations.

Tab. 2 exemplifies HyperLogLog structures with two buckets applied to the example dataset from Tab. 1. Note that in practical scenarios, more buckets should be used to enhance HyperLogLog's accuracy. Given that all column pairs are IND candidates, the above described IND test identifies all actual INDs correctly. For instance, *word1* $\subseteq$ *word* is correctly deemed to be an IND, because the HyperLogLog bucket values of *word2*, i.e., 1 and 3, are less then or equal to the respective bucket values of *word*, which are also 1 and 3. This result completeness is guarantueed, because if $X \subseteq Y$ is actually an IND, then all values of X must be considered in the HyperLogLog structure of Y. Correctness of the result cannot be guaranteed, though. For instance, *word* $\subseteq$ *word1* is deemed to be an IND judging from the HyperLogLog structures, although it is actually not. As mentioned above, HyperLogLog is not well-suited to compare columns with only few distinct values. Therefore, we complement it with a second IND test as presented in the next section.

Prefix	**word**	**lang**	**type**	**word1**	**lang1**	**type1**	**word2**	**lang2**	**type2**	**fit**
0	1	1	0	1	1	0	0	0	0	0
1	3	1	2	3	0	1	1	1	1	0

Tab. 2: HyperLogLog structures for single columns of the example dataset.

4.3 Hybrid Inclusion Dependency Test

HyperLogLog is a stochastic counting approximation that works particularly well for IND tests where both column combinations have many distinct values. To fill its blind spot – IND candidates containing a column combination with only few values – the IND tests additionally use an inverted index. The IND testing with inverted indexes has first been introduced by De Marchi et al. [DMLP09]. The basic idea is to build an inverted index of the input dataset that maps each value to the columns it appears in. For instance, for our example dataset in Tab. 1 such an inverted index maps the value *en* to the set of columns {*lang*, *lang1*, *lang2*}. Now to find all columns that include a certain column X, it suffices to select all column sets that contain X and intersect them. Applying this procedure for every column X yields all the INDs in the dataset.

To avoid scaling problems, Faida cannot create such an inverted index on the entire dataset. However, it is possible to create such an inverted index for a subset of the values (or rather a subset of the hashes as described in Sect. 4.1) in the dataset and apply the said IND test to it. This idea seems promising, because we can control the sample size and yet focus the sample in such a manner that it comprises especially the values of columns with only few distinct values (cf. Sect. 4.1), i.e., those cases where HyperLogLog is not so well-suited. Moreover, this approach still preserves result completeness: If $X \subseteq Y$ is an IND, then $\sigma(X) \subseteq \sigma(Y)$ has to hold, too, where σ selects only those values that are in the sample.

Algorithm 2 implements this idea. It starts by taking the sample tuples for each table (cf. Sect. 4.1). Then, it creates an inverted index for all column combinations that appear

in any of the IND candidates (Lines 1–7). Next, Algorithm 2 builds a HyperLogLog structure for each column combinations. Afterwards, the algorithm needs to initialize a flag in *isCovered* to keep track of whether all values for a certain column combination are actually found in the sample and, thus, in the inverted index (Lines 8–12). Having initialized all relevant data structures, the algorithm iterates all values of all column combinations (Line 13). If a value is included in the table samples and, thus, a key of the inverted index, the corresponding index entry is updated with the column combination of that value (Lines 14–16); otherwise, the algorithm updates the corresponding HyperLogLog structure with that value (Line 17). In the latter case, the algorithm also notes that the respective column combination is not completely covered by the inverted index (Line 19). Whether or not a column combination is covered becomes relevant in the subsequent phase where the IND candidates are actually tested. In the beginning of that phase, only those IND candidates are retained that hold on the inverted index (Line 20). Now, if the dependent

Algorithm 2: Hybrid IND test

Input : set of IND candidates I_c
samples of hashed tuples for each table $\mathcal{T}_s$
hashes for all value combinations V
Output verified INDs I
:

```
1  invertedIndex ← mapping(defaultValue = ∅);
2  foreach T_s ∈ 𝒯_s do
3      C ← relevantColumnCombinations(T_s, I);
4      foreach t ∈ T_s do
5          foreach c ∈ C do
6              v ← t[c];
7              invertedIndex[v] ← invertedIndex[v] ∪ {c}
8  isCovered ← mapping();
9  hlls ← mapping();
10 foreach c ∈ allColumnCombinations(I_c) do
11     isCovered[c] ← true;
12     hlls[c] ← hyperLogLog();
13 foreach v ∈ V do
14     c ← columnCombination(v);
15     C ← invertedIndex[v];
16     if C ≠ ∅ then invertedIndex[v] ← C ∪ {c};
17     else
18         insert(v into hlls[c]);
19         isCovered[v] ← false;
20 I'_c ← testAll(I_c on invertedIndex);
21 foreach ⟨X ⊆ Y⟩ ∈ I'_c do
22     if isCovered[X] ∨ (¬isCovered[Y] ∧ test(⟨X ⊆ Y⟩ on hll[X] and hll[Y])) then
23         I ← I ∪ {⟨X ⊆ Y⟩};
```

column combination X of any retained IND candidate $X \subseteq Y$ is covered by the inverted index, we can directly promote it as a valid IND (Lines 21–23); otherwise, if neither X nor Y is covered, we additionally perform the HyperLogLog-based IND test to verify the candidate. Note that, if Y is covered but X is not, then there must be some value in X that violates $X \subseteq Y$. Thus, we do not add it to the set of actual INDs I in that case.

In summary, the presented IND test follows a hybrid strategy using HyperLogLog and an inverted index. The sampling-based inverted index reliably discerns INDs between column combinations with only few distinct values. If, in contrast, IND candidates between column combinations with many distinct values need to be tested, it automatically switches to the stochastic HyperLogLog-based test that scales well because of its constant memory footprint regardless of the size of the input data. Still, the inverted index reinforces the test as a "control sample".

5 Scrap Inclusion Dependencies

Exact IND discovery algorithms, and also Faida, deliberately exclude some uninteresting INDs in their result sets, even though the respective dataset actually satisfies them. Those INDs have certain *syntactical* properties: First, there are *trivial* INDs $X \subseteq X$ with equal dependent and referenced column combinations, which always hold. Second, discovery algorithms respect permutability of INDs, i.e., if $AB \subseteq CD$ is a valid IND, then $BA \subseteq DC$ must also hold. Thus, it is sufficient to check (and report) only a single IND candidate from such permutation classes. Omitting these two kinds of INDs reduces both the amount of resources needed during the discovery process and the size of the output that usually needs to undergo further, often manual, processing.

On the face of those benefits, we propose to extend the criteria for omissible INDs from syntactical properties to instance-based properties, i.e., properties of the data comprised in the columns of an INDs. Specifically, we argue that columns that contain only NULL values (which we call *NULL columns*) and columns that contain only a single distinct value (which we call *constant columns*) are only contained in INDs that are degenerate and not actually useful for typical IND use-cases, such as those described in Sect. 1. Thus, it is fair to omit those *scrap INDs* – and it is also significant, because in our experiments we found scrap INDs to appear quite frequently.

NULL columns. There are several ways to interpret NULL values, e.g., using possible-world semantics or simply treating it as another domain value [Kö16]. Another approach is to treat NULLs as "no value", which conforms to the semantics of foreign keys in SQL. Under that interpretation, a column with only NULLs basically contains no values at all; its value set is the empty set. Because the empty set is a subset of all other sets, for a NULL column A and *any* other column B, $A \subseteq B$ is a valid IND. However, not describing an actual inclusion of values, this IND is unlikely useful. Furthermore, any other IND $X \subseteq Y$ can be extended to $XA \subseteq YB$, where A and X, as well as B and Y lie in the same respective tables. Again, this extension is not useful, because it does not refine $X \subseteq Y$, i.e., AX does not discern tuples beyond X. NULL columns are a common phenomenon. They can occur

when schemata provide overly detailed column sets or simply when the data for a column cannot be ascertained. Therefore, FAIDA detects NULL columns during the preprocessing (cf. Sect. 4.1), removes them from candidate generation, and reports them in the end. In this way, no INDs involving NULL columns will be discovered and the user is informed why.

Constant columns. We call a column constant if it stores the same non-null value for every tuple. During the analysis of several real-world datasets, we found that in all cases of such constant columns, the value in question (e.g., "−1" or an empty string) either is a surrogate for a NULL value or a default value (in the sense of SQL's `DEFAULT` keyword). Arguably, INDs containing constant columns are omissible: If the constant is a NULL surrogate, then the same rationale as for NULL columns applies; in any other case, constant columns still do not provide much value, because they do not discern the tuples of their table. Such INDs with constant columns can bloat the IND search and result space. In particular, two constant columns A and B with the same value can be added to any IND $X \subseteq Y$ and form the valid IND $XA \subseteq YB$ where A and X as well as B and Y are from the same table. Thus, FAIDA also detects constant columns in order to report and remove them.

By excluding the two described kinds of scrap INDs, FAIDA often gains significant performance improvements and, at the same time, enhances the quality of the discovered INDs. Note that the removal of scrap INDs is an additional, optional improvement of FAIDA and not a necessity to run the algorithm.

6 Evaluation

In our evaluation, we demonstrate both the efficiency and effectiveness of FAIDA. Regarding efficiency, we want to answer two main questions: (i) *How does FAIDA compare to an exact state-of-the-art IND discovery algorithm, namely BINDER?* (ii) *How well does FAIDA scale to large datasets?* To investigate the effectiveness, we address the following questions: (iii) *How good is FAIDA's result quality and to what extent is it influenced by its parameterization?* (iv) *What are the effects of omitting the scrap INDs?* We first briefly describe our experimental setup and then answer these questions in various experiments.

6.1 Experimental setup

Hardware. All experiments were run on a machine with an Intel Core i5-4690 CPU with 1600 MHz, 8 GB of main memory, and a Seagate Barracuda ST3000DM001 3 TB hard disk. We used Ubuntu 14 and the Oracle JRE 1.8u45 with a maximum 6 GB heap size.

Datasets. The datasets used for evaluation are all publicly available. Some details about those datasets are listed on the left-hand side of Tab. 3. Further information and links for all datasets, as well as an implementation of FAIDA, can be found at `https://hpi.de/naumann/projects/repeatability/data-profiling/metanome-ind-algorithms.html`.

Parameterization. FAIDA is configured via two parameters: The sampling-based inverted index requests the number of values to sample from each column, and the HYPERLOGLOG structures require a desired accuracy of their count estimates, which effectively designates their number of buckets. While FAIDA is guaranteed to find all INDs, it potentially reports incorrect INDs due to its approximative nature. Thus, the configuration of the two parameters impacts FAIDA's output quality: larger samples and more HYPERLOGLOG counters reveal violations in IND candidates more accurately. In our experiments, we set the sampling parameter to a default of 500 and the HYPERLOGLOG accuracy to a default of 0.1 %, which roughly allocates 640 KiB of main memory for 1,000,000 buckets per HYPERLOGLOG structure. Sect. 6.4 investigates FAIDA's sensitivity w.r.t. this parameterization and shows that our defaults are a rather conservative and robust choice that incur no or only very few false positive INDs. Thus, our defaults are a reasonable choice for the following comparison with BINDER.

6.2 Comparison of FAIDA and BINDER

The premise of approximate IND discovery is that a little loss in result quality can be traded for large performance improvements. Even though FAIDA always discovered exactly the correct INDs in our experiments, it relinquishes correctness guarantees, and, in turn, it should be more efficient than exact IND discovery algorithms. To verify this, we compare FAIDA's runtimes on various datasets with those of the state-of-the-art algorithm for exact IND discovery, BINDER [Pa15]. Note that FAIDA prunes scrap INDs, as introduced in Sect. 5. This novel pruning technique is not restricted to approximate IND discovery and can be applied to other IND discovery algorithms as well. To allow a fair comparison, we modified BINDER to also prune scrap INDs and provide a separate evaluation of the scrap IND pruning in Sect. 6.5.

In addition, we considered simple approximate IND discovery baselines. To determine the impact of using hashes rather than actual value combinations, we modified BINDER to hash long value combinations and operate on those hashes then. This modification always was approximately 20 % slower, because the additional hashing costs could not be redeemed. Also, we considered FAIDA without its inverted index, thereby detecting INDs solely using HYPERLOGLOG. However, while the performance overhead of the inverted index is small, leaving it out often causes false positive INDs. Those yield unnecessary IND candidates, so that eventually performance declines (see Sect. 6.4). With these modifications being inferior, we focus only on FAIDA and BINDER in the following.

Tab. 3 shows the results and runtimes of both algorithms to discover the INDs in various datasets. FAIDA outperforms BINDER consistently by a factor of five to six. Both algorithms generated and tested exactly the same IND candidates, which means that FAIDA's data preprocessing and approximate IND tests are more efficient than BINDER's exact, hash partition-based IND test.

The reason for this improvement is two-fold. At first, FAIDA tests INDs using compact hashes rather than the actual values from the datasets, which allows for more efficient comparisons

and reduces memory requirements. For instance, the TESMA dataset contains a lot of long string values. Although, this dataset contains only unary INDs, FAIDA's hashing approach can drastically reduce the computation load and easily redeems its data preprocessing overhead. In addition to that, value combinations of n-ary IND candidates often become quite long. Again, FAIDA represents those by a single hash value. The second reason for the performance improvement is found in the fact that FAIDA summarizes large datasets with small HYPERLOGLOG structures and does not need to do any out-of-core execution. BINDER, in contrast, needs to spill data to disk when processing large datasets. The I/O effort can slow it down severely – in particular for long values and value combinations, respectively.

Dataset	Size	Non-constant columns	Non-scrap n-ary INDs	Max. arity	Runtime BINDER	Runtime FAIDA
CENSUS	117 MB	48	222	6	39 sec	6 sec
WIKIRANK	730 MB	25	118	6	2 min 44 sec	26 sec
TESMA	1.2 GB	114	2	1	1 min 36 sec	25 sec
TCP-H 70	79.4 GB	60	111	3	9 h 32 min	1 h 47 min

Tab. 3: Comparative evaluation for n-ary IND detection.

6.3 Scalability

On the face of ever-growing datasets, scalability is an important property of IND discovery algorithms. In particular, we investigate two scalability dimensions, namely the number of rows and the numbers of columns in a dataset, and compare FAIDA with BINDER along these dimensions.

Row Scalability. To analyze the row scalability of FAIDA, it makes sense to reduce the impact of other factors affecting its runtime, such as value distributions and the number of INDs among the test datasets. To keep those other impact factors steady across datasets of different size, we use the TPC-H dataset generator to create datasets with varying numbers of rows but with the same schema and the same foreign keys. Nevertheless, we observed a few more, likely spurious, INDs, as the randomly generated data volume increases. Because their number is very small, they hardly affect runtime, though: TCP-H 1 has 104 INDs while TCP-H 100 has 113.

Fig. 3 displays the results of the row scalability experiment for FAIDA and BINDER. While both algorithms exhibit a linear scaling behavior, FAIDA is always around five times faster than BINDER. In other words, the larger the dataset is, the greater are the absolute time savings of FAIDA compared to BINDER.

Column Scalability. To test the runtime behavior with regard to the number of columns, we used a subset of the PDB dataset, namely 15 tables with at least 20 columns each. Then, we executed FAIDA and BINDER 20 times on those tables, thereby only taking into account the first k columns for each $1 \leq k \leq 20$. Incrementing the number of considered columns

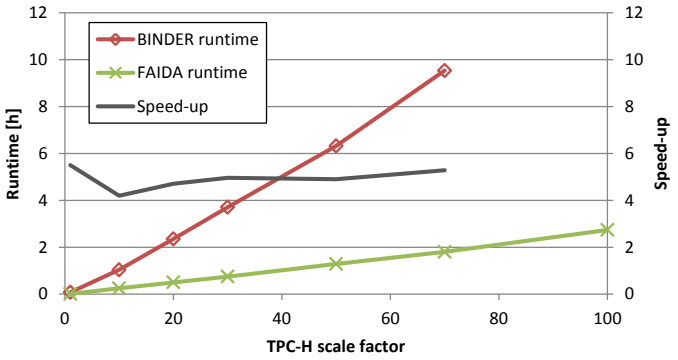


Fig. 3: Runtime scaling of BINDER and FAIDA with the number of rows.

in each table, rather than just incrementing the number of considered tables, mitigates the runtime impact of varying numbers of tuples in the tables and yields a smooth increase of the processed data volume.

Fig. 4 shows the runtime of FAIDA and BINDER together with the number and distribution of discovered INDs. Apparently, both algorithms scale somewhat linearly w.r.t. the number of INDs. Nevertheless, FAIDA scales a lot better in the presence of n-ary INDs, where its approximation schemes take particular effect: At first, FAIDA resorts to its hashed column store to test IND candidates, while BINDER has to re-read the complete input dataset multiple times to test IND candidates of different arities. Second, FAIDA works exclusively on compact hashes; BINDER in contrast concatenates values and shuffles the larger value combinations to test n-ary IND candidates. Finally, FAIDA's HYPERLOGLOG structures keep its memory footprint relatively small, while BINDER at some point needs to spill the mentioned value combinations to disk in order to shuffle them. This spilling causes BINDER's drastic runtime increase for more than 250 columns. All of the above experiments demonstrate that FAIDA trades result correctness for considerable performance gains.

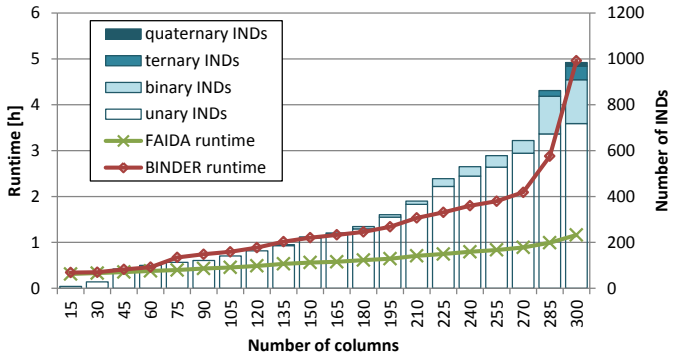


Fig. 4: Runtime scaling of BINDER and FAIDA with the number of columns.

6.4 Result Correctness

FAIDA uses two approximate data structures to test IND candidates: a sampling-based inverted index and HYPERLOGLOG structures, both of which can trade main memory requirements for accuracy. Hence, it is important to size them in a way that lets FAIDA yield accurate results without straining main memory too much. To explore this trade-off, we executed FAIDA with different HYPERLOGLOG accuracies (see Sect. 4.2) and column sample sizes (see Sect. 4.3), thereby measuring the *false-positive rate*, i.e., the ratio of incorrectly reported INDs.

Tab. 4 displays the *maximum* false-positive rate of FAIDA across the seven datasets COMA, CENSUS, BIOSQLSP, WIKIRANK, CATH, TESMA, and TPC-H[3], and reveals two interesting insights. First, it is clearly visible that the sampling-based inverted index and the HYPERLOGLOG structures complement one another. In particular, the HYPERLOGLOG structures alone did not always yield exact results and the inverted index has to be quite large to achieve full correctness. However, in combination, the two data structures exhibit superior performance. As a second insight, it becomes apparent that a reasonably sized inverted index and HYPERLOGLOG structures can robustly provide exact IND results. As a matter of fact, the column sample size of 500 and the HYPERLOGLOG accuracy of 0.1 % that we used in our efficiency experiments turn out to be a rather conservative choice. FAIDA is quite robust with respect to parameter settings. While these results, of course, do not imply that it will discover exactly the correct INDs on any given dataset, they do indicate a high confidence in FAIDA's results.

	HLL accuracy		
Sample size	10 %	1 %	0.1 %
1	6.000	0.082	0.024
10	0.243	0.047	0.012
100	0.094	**0.000**	**0.000**
1,000	0.036	0.000	0.0000
10,000	**0.000**	0.000	0.0000

Tab. 4: Maximum false positive rate of FAIDA over various datasets under different parameterizations.

Having shown that the combination of a sampled inverted index and HYPERLOGLOG yields high precision, it is intriguing to investigate how FAIDA behaves when HYPERLOGLOG is replaced with other data summarization techniques. For this purpose, we repeated the above experiment with bottom-k sketches, as proposed in [Zh10], and with Bloom filters. To make these techniques comparable to HYPERLOGLOG, we configured the size of the Bloom filter and the number hashes in the bottom-k sketch, respectively, such that they consume as much main memory as HYPERLOGLOG for the various accuracy settings from Tab. 4. We found that bottom-k sketches are not a good choice: Although still yielding good results, bottom-k sketches performed *at most* as well as (but often worse than) HYPERLOGLOG and Bloom

[3] See `https://hpi.de/naumann/projects/repeatability/data-profiling/metanome-ind-algorithms.html` for downloads and details of these datasets.

filters under all parameterizations. This is because bottom-k sketches do not partition the hash space, which would allow a pairwise comparison of their hash values, as is the case for bits in a Bloom filter or buckets in a HyperLogLog structure. However, we also found that Bloom filters performed similarly well as HyperLogLog and are an eligible replacement.

6.5 Omitting scrap INDs

Scrap INDs are those INDs that involve either NULL columns (columns containing no values other than NULL) and/or constant columns (columns containing only a single value). In Sect. 5 we argue that such INDs are not meaningful for the typical IND-based applications and ignoring them could save much computation. It remains to show that the class of scrap INDs is common and its dedicated treatment worthwhile.

To this end, we analyzed the different types of INDs in various datasets. The results are displayed in Fig. 5. Approximately two thirds of all INDs in this experiment are scrap INDs. While the majority of scrap INDs involve NULL columns, we observe that datasets can also comprise many scrap INDs related to constant columns, such as ENSEMBL. Furthermore, we measured the runtime of Faida with and without pruning of scrap INDs and found it to be beneficial. While for three out of the seven datasets, performance was not affected, for the other four datasets, the pruning indeed yielded a performance improvement. On the EMDE dataset, particularly, we observed a speed-up of factor 20. In consequence, it seems appropriate to detect and prune scrap INDs already during the IND discovery process.

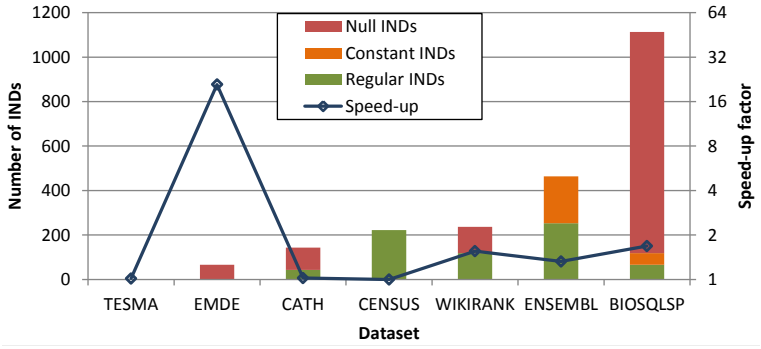


Fig. 5: Break-down of IND types for various datasets.

7 Conclusion

We presented Faida, an approximate algorithm for the n-ary IND discovery problem. Faida uses a symbiotic combination of data preprocessing, hashes, a sampling-based inverted index, and HyperLogLog to test INDs in a highly efficient and scalable manner. In our experiments, we found our algorithm to be as much as six times faster than the exact state-of-the-art IND discovery algorithm Binder. Besides performance aspects, Faida further guarantees result completeness, i.e., it will find all INDs in a given dataset.

Although incorrect INDs might be reported, FAIDA did not yield any false positives in our experiments. This shows the effectiveness of our hybrid IND test. A promising direction for future research is to adapt FAIDA for incremental IND discovery: With the low memory footprint of its data structures, FAIDA might be a particularly good fit to maintain a set of INDs on evolving, dynamic datasets. However, especially updating those data structures on value deletions or changes is a challenging task.

Acknowledgements. This research was partially funded by the German Research Society (DFG grant no. FOR 1306).

References

[AGN15] Abedjan, Ziawasch; Golab, Lukasz; Naumann, Felix: Profiling relational data: a survey. VLDB Journal, 24(4):557–581, 2015.

[Ba07] Bauckmann, Jana; Leser, Ulf; Naumann, Felix; Tietz, Véronique: Efficiently detecting inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE). pp. 1448–1450, 2007.

[Bo05] Bohannon, Philip; Fan, Wenfei; Flaster, Michael; Rastogi, Rajeev: A cost-based model and effective heuristic for repairing constraints by value modification. In: Proceedings of the International Conference on Management of Data (SIGMOD). pp. 143–154, 2005.

[CTF88] Casanova, Marco A.; Tucherman, Luiz; Furtado, Antonio L.: Enforcing Inclusion Dependencies and Referencial Integrity. In: Proceedings of the International Conference on Very Large Databases (VLDB). pp. 38–49, 1988.

[DMLP09] De Marchi, Fabien; Lopes, Stéphane; Petit, Jean-Marc: Unary and n-ary inclusion dependency discovery in relational databases. Journal of Intelligent Information Systems, 32(1):53–73, 2009.

[DMP03] De Marchi, Fabien; Petit, Jean-Marc: Zigzag: a new algorithm for mining large inclusion dependencies in databases. In: Proceedings of the International Conference on Data Mining (ICDM). pp. 27–34, 2003.

[Fl07] Flajolet, Philippe; Fusy, Éric; Gandouet, Olivier; Meunier, Frédéric: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: Proceedings of the International Conference on Analysis of Algorithms (AofA). pp. 127–146, 2007.

[Gr98] Gryz, Jarek: Query folding with inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE). pp. 126–133, 1998.

[Kö16] Köhler, Henning; Leck, Uwe; Link, Sebastian; Zhou, Xiaofang: Possible and certain keys for SQL. VLDB Journal, 25(4):571–596, 2016.

[KPN15] Kruse, Sebastian; Papenbrock, Thorsten; Naumann, Felix: Scaling out the discovery of inclusion dependencies. In: Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW). pp. 445–454, 2015.

[KR03] Koeller, Andreas; Rundensteiner, Elke: Discovery of high-dimensional inclusion dependencies. In: Proceedings of the International Conference on Data Engineering (ICDE). pp. 683–685, 2003.

[LPT02] Lopes, Stéphane; Petit, Jean-Marc; Toumani, Farouk: Discovering interesting inclusion dependencies: application to logical database tuning. Information Systems, 27(1):1–19, 2002.

[Pa15] Papenbrock, Thorsten; Kruse, Sebastian; Quiané-Ruiz, Jorge-Arnulfo; Naumann, Felix: Divide & conquer-based inclusion dependency discovery. Proceedings of the VLDB Endowment (PVLDB), 8(7):774–785, 2015.

[Ro09] Rostin, Alexandra; Albrecht, Oliver; Bauckmann, Jana; Naumann, Felix; Leser, Ulf: A Machine Learning Approach to Foreign Key Discovery. In: Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB). 2009.

[SM16] Shaabani, Nuhad; Meinel, Christoph: Detecting Maximum Inclusion Dependencies without Candidate Generation. In: Database and Expert Systems Applications (DEXA). pp. 118–133, 2016.

[Zh10] Zhang, Meihui; Hadjieleftheriou, Marios; Ooi, Beng Chin; Procopiuc, Cecilia M; Srivastava, Divesh: On multi-column foreign key discovery. Proceedings of the VLDB Endowment (PVLDB), 3(1-2):805–814, 2010.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 227

EFM-DBSCAN: Ein baumbasierter Clusteringalgorithmus unter Ausnutzung erweiterter Leader-Umgebungen

Philipp Egert[1]

Abstract:

DBSCAN ist ein dichte-basierter Clusteringalgorithmus, der beliebig geformte Cluster erkennt und sie von Rauschen trennt. Aufgrund der Laufzeit von $O(n^2)$ ist seine Anwendung jedoch auf kleine Datenkollektionen beschränkt. Um diesen Aufwand zu reduzieren, wurde der auf dem Konzept der Leader-Umgebung basierende Algorithmus FM-DBSCAN vorgestellt, der für beliebige Metriken dasselbe Clustering wie DBSCAN liefert. In dieser Arbeit wird nun basierend auf FM-DBSCAN das Verfahren EFM-DBSCAN entwickelt. EFM-DBSCAN nutzt die folgenden zwei Konzepte zur Effizienzsteigerung: (a) eine baumbasierte Partitionierung und (b) die Erweiterung der Objekte einer Leader-Umgebung um die Distanzen zu ihrem Leader. Erste Experimente zeigen, dass EFM-DBSCAN bis zu einem Faktor 17 weniger Distanzberechnungen und bis zu einem Faktor 13 weniger Rechenzeit als FM-DBSCAN benötigt. Gegenüber DBSCAN wurde ein Faktor von bis zu 10^4 eingespart.

Keywords: Density-based Clustering, DBSCAN, Leaders, Leader Neighbourhood

1 Einleitung

Dichte-basierte Clusteringverfahren sind häufig genutzte Algorithmen zum Clustern von Objekten. Sie sind in der Lage, beliebig geformte Cluster aufzufinden und diese von Rauschen zu trennen. DBSCAN [Es96] ist ein Vorreiter dieser Verfahren, der aufgrund seines eleganten Algorithmus und der nicht benötigten Vorgabe der Clusteranzahl für viele Einsatzgebiete attraktiv ist. Seine Laufzeit von $O(n^2)$ ist jedoch oft nicht praktikabel.

Zur Aufwandsverringerung wurden mehrere Ansätze entwickelt, unter anderem FM-DBSCAN [Eg16]. FM-DBSCAN ist ein exaktes Verfahren, dass für eine beliebige Metrik dasselbe Clustering wie DBSCAN liefert. Hierzu werden die Objekte in Leader-Umgebungen partitioniert und anschließend geclustert, indem die Erreichbarkeitsbegriffe von DBSCAN auf Leader-Umgebungen übertragen werden. Die Struktur der Leader-Umgebungen und die Ausnutzung der Dreiecksungleichung ermöglichen eine Beschleunigung des Clusterns. Zwar ist die Laufzeit immer noch $O(n^2)$, jedoch kann die Rechenzeit, aufgrund der Reduzierung der Anzahl der Distanzberechnungen, oft drastisch gesenkt werden.

In dieser Arbeit wird, basierend auf FM-DBSCAN, EFM-DBSCAN mit den folgenden neuen Konzepten entwickelt, um die Anzahl der Distanzberechnungen noch weiter zu verringern: (a) Eine baumbasierte Partitionierung zur Beschleunigung der Berechnung der

[1] Brandenburgische Technische Universität Cottbus–Senftenberg, Institut für Informatik, Informations- und Medientechnik, Konrad-Wachsmann-Allee 5, 03046 Cottbus, egertphi@b-tu.de

Leader-Partition und (b) die Erweiterung der Objekte einer Leader-Umgebung um die Distanzen zu ihrem Leader für schärfere Ausschlussbedingungen. Die Ausschlussbedingungen ermöglichen gleichzeitig eine effektivere Ausschlussstrategie anhand schrumpfender Leader-Umgebungen, um den Test der direkten Dichte-Erreichbarkeit zweier Leader-Umgebungen und den Test auf die Kernobjekteigenschaft zu beschleunigen.

2 Stand der Technik

In diesem Abschnitt wird der Stand der Technik vorgestellt. Hierzu werden einleitend die Algorithmen DBSCAN und FM-DBSCAN kurz vorgestellt. Abschließend werden noch weitere, aus der Literatur bekannte, Ansätze zur Beschleunigung von DBSCAN dargelegt.

2.1 DBSCAN

DBSCAN ist ein dichte-basiertes Clusteringverfahren von Ester et al. [Es96]. Ausgangspunkt ist eine endliche Menge von Objekten $O \subseteq \mathbb{U}$, wobei $\mathbb{U}$ das Universum aller möglichen Objekte ist. Über $\mathbb{U}$ sei eine Metrik d zum Vergleich der Objekte definiert. Die wesentlichen Definitionen zur Beschreibung eines Clusters werden anschließend eingeführt. Alle Definitionen sind dabei abhängig von den DBSCAN-Parametern ϵ und *minPts*.

Definition 1 (Kernobjekt)**.** Ein Objekt $o \in O$ heißt *Kernobjekt*, wenn $|N_\epsilon(o)| \geq minPts$ gilt, mit $N_\epsilon(o) = \{\hat{o} \in O \mid d(o, \hat{o}) \leq \epsilon\}$. Gilt dies nicht, heißt *o Nicht-Kernobjekt*.

Definition 2 (Direkte Dichte-Erreichbarkeit)**.** Ein Objekt $p \in O$ ist *direkt dichte-erreichbar* von $q \in O$, wenn q ein Kernobjekt und $p \in N_\epsilon(q)$ ist.

Definition 3 (Dichte-Erreichbarkeit)**.** Ein Objekt $p \in O$ ist *dichte-erreichbar* von einem Objekt $q \in O$, wenn es eine Folge von Objekten $p_1, \ldots, p_m \in O$ gibt, sodas $p_1 = q, p_m = p$ ist und es gilt: p_{i+1} ist direkt dichte-erreichbar von p_i für $1 \leq i < m$.

In DBSCAN bestimmt man einen Cluster C nun dadurch, dass man für ein Kernobjekt $o \in O$ alle dichte-erreichbaren Objekte iterativ einsammelt. Hierfür werden von o alle direkt dichte-erreichbaren Objekte bestimmt, also $N_\epsilon(o)$, und C hinzugefügt. Anschließend werden von allen Kernobjekten aus $N_\epsilon(o)$ deren direkt dichte-erreichbaren Objekte bestimmt und C beigefügt. Diese Schritte werden fortgesetzt, bis C nicht mehr erweitert werden kann. Alle Objekte die zu keinem Cluster gehören, werden als *Rauschen* bezeichnet.

2.2 FM-DBSCAN

FM-DBSCAN ist ein Clusteringverfahren, welches für eine beliebige Metrik dasselbe Clustering wie DBSCAN berechnet, jedoch effizienter ist [Eg16]. Nachfolgend werden die essenziellen Begriffe von FM-DBSCAN eingeführt und der Algorithmus erläutert.

Definition 4 (Leader-Umgebung)**.** Ein Tupel $(l, N_\epsilon^L(l))$ ist eine *Leader-Umgebung* in O, wenn $N_\epsilon^L(l) \subseteq N_\epsilon(l)$ mit $l \in N_\epsilon^L(l)$ ist. l ist der Repräsentant von $N_\epsilon^L(l)$ und heißt *Leader*.

Definition 5 (Leader-Partition)**.** Eine *Leader-Partition* von O ist eine Menge von Leader-Umgebungen $L = \{(l_1, N_\epsilon^L(l_1)), \ldots, (l_m, N_\epsilon^L(l_m))\}$ mit $d(l_i, l_j) > \epsilon$, $N_\epsilon^L(l_i) \cap N_\epsilon^L(l_j) = \emptyset$ für $i \neq j$ und $\bigcup_{i=1}^{m} N_\epsilon^L(l_i) = O$.

Definition 6 (Direkte Dichte-Erreichbarkeit)**.** Eine Leader-Umgebung $(l_p, N_\epsilon^L(l_p))$ ist *direkt dichte-erreichbar* von einer Leader-Umgebung $(l_q, N_\epsilon^L(l_q))$ in O, wenn es eine Folge von Objekten $p_1, \ldots, p_m \in (N_\epsilon^L(l_p) \cup N_\epsilon^L(l_q))$ gibt, mit $p_1 = l_q, p_m = l_p$ und es gilt: p_{i+1} ist direkt dichte-erreichbar von p_i für $1 \leq i < m$, und l_p ist ein Kernobjekt in O oder $|N_\epsilon^L(l_p)| = 1$.

Aus der Definition 6 folgt direkt: Ist $(l_p, N_\epsilon^L(l_p))$ direkt dichte-erreichbar von $(l_q, N_\epsilon^L(l_q))$, so sind alle Objekte $o \in (N_\epsilon^L(l_q) \cup N_\epsilon^L(l_p))$ dichte-erreichbar von l_q und gehören somit zum selben Cluster. Die Dichte-Erreichbarkeit wird analog zur Dichte-Erreichbarkeit von Objekten definiert. Im Anschluss werden noch drei Sätze vorgestellt, die es FM-DBSCAN ermöglichen, erheblichen Berechnungsaufwand im Vergleich zu DBSCAN einzusparen.[2]

Satz 1. Seien $(l_p, N_\epsilon^L(l_p))$ und $(l_q, N_\epsilon^L(l_q))$ zwei Leader-Umgebungen. $(l_p, N_\epsilon^L(l_p))$ ist direkt dichte-erreichbar von $(l_q, N_\epsilon^L(l_q))$ in O, genau dann, wenn l_q ein Kernobjekt ist und mindestens einer der folgenden beiden Fälle eintritt:
1. Es gibt zwei Objekte $p \in N_\epsilon^L(l_p), q \in N_\epsilon^L(l_q)$ mit $d(p, q) \leq \epsilon$ und p, q, l_p sind Kernobjekte.
2. $|N_\epsilon^L(l_p)| = 1$ und es gibt ein Kernobjekt $q \in N_\epsilon^L(l_q)$ in O mit $d(l_p, q) \leq \epsilon$.

Satz 2. Seien $p, q \in O$. Ist $d(q, p) > 2\epsilon$, so gilt: $\forall o \in N_\epsilon(p) : d(q, o) > \epsilon$.

Satz 3. Seien $p, q \in O$. Ist $d(p, q) > 3\epsilon$, so gilt: $\forall o_1 \in N_\epsilon(p), o_2 \in N_\epsilon(q) : d(o_1, o_2) > \epsilon$.

Der Algorithmus von FM-DBSCAN unterteilt sich in zwei Phasen, die Berechnung der Leader-Partition und das anschließende Clustern mithilfe der Leader-Partition.

Berechnung der Leader-Partition: Die Leader-Partition wird iterativ aufgebaut, indem jedes Objekt $o \in O$ der ersten Leader-Umgebung $(l, N_\epsilon^L(l))$ zugewiesen wird, für die $d(o, l) \leq \epsilon$ gilt. Gibt es eine solche Leader-Umgebung nicht, so wird die neue Leader-Umgebung $(o, \{o\})$ erzeugt. Die Überprüfungsreihenfolge der Leader-Umgebungen ist durch deren Erzeugungsreihenfolge festgelegt, weswegen wir diese Partitionierung auch First-come, first-served (FCFS)-Partitionierung nennen.

Clustern anhand der Leader-Partition: Die Bestimmung eines Clusters C ist analog zu DBSCAN. Ausgehend von einer Leader-Umgebung $(l, N_\epsilon^L(l))$, dessen Leader l ein Kernobjekt ist, werden alle dichte erreichbaren Leader-Umgebungen iterativ eingesammelt. Der wesentlicher Unterschied ist die Ermittlung der direkt dichte-erreichbaren Leader-Umgebungen. Hierfür bestimmt man mit Satz 3 zuerst die potentiell möglichen direkt dichte-erreichbaren Leader-Umgebungen (Kandidaten) und testet anschließend für jeden Kandidaten $(l_c, N_\epsilon^L(l_c))$, ob er wirklich direkt dichte-erreichbar von $(l, N_\epsilon^L(l))$ ist. Die maßgeblich Kosten entstehen durch die Tests der Kernobjekteigenschaft und der direkten Dichte-Erreichbarkeit. Nachfolgend werden beide Methoden beschrieben, da in EFM-DBSCAN dort wesentliche Änderungen erfolgen.

[2] Die Beweise sind aus Platzgründen in http://dbis.informatik.tu-cottbus.de/down/pdf/Leader_Umgebungen.pdf zu finden.

isCoreObject überprüft, ob das Objekt o ein Kernobjekt ist oder nicht, indem sie über die Objekte o_2 aller Leader-Umgebungen iteriert und zählt (Variable *counter*), wie oft $d(o, o_2) \leq \epsilon$ erfüllt ist. Zum Ausschluss von Leader-Umgebungen wird Satz 2 herangezogen. Handelt es sich bei o um einen Leader, so wird *counter* mit $|N_\epsilon^L(o)|$ initialisiert und $(o, N_\epsilon^L(o))$ von der weiteren Suche ausgeschlossen. Ein frühzeitiger Abbruch erfolgt, wenn $counter \geq minPts$ ist. Anhand eines im Objekt o gespeicherten Attributes kann das mehrmalige Berechnen der Kernobjekteigenschaft verhindert werden.

isDirectDensityReachable überprüft für $(l_1, N_\epsilon^L(l_1))$ und $(l_2, N_\epsilon^L(l_2))$, ob $(l_2, N_\epsilon^L(l_2))$ direkt dichte-erreichbar von $(l_1, N_\epsilon^L(l_1))$ ist. Dabei wird davon ausgegangen, dass l_1 schon als Kernobjekt erkannt wurde. Zuerst wird getestet, ob $(l_2, N_\epsilon^L(l_2))$ direkt dichte-erreichbar von $(l_1, N_\epsilon^L(l_1))$ sein kann (l_2 ist ein Kernobjekt oder $|N_\epsilon^L(l_1)| = 1$). Die eigentliche Überprüfung wird durch zwei ineinander verschachtelte Schleifen realisiert, sodass im schlimmsten Fall alle Objekte aus $N_\epsilon^L(l_1)$ mit denen aus $N_\epsilon^L(l_2)$ verglichen werden. Um dies zu verhindern, wird der Satz 2 angewendet. Haben zwei Objekte $o_1 \in N_\epsilon^L(l_1)$ und $o_2 \in N_\epsilon^L(l_2)$ eine Distanz $d(o_1, o_2) \leq \epsilon$, so wird der Satz 1 überprüft und ggf. erfolgreich abgebrochen.

2.3 Andere Arbeiten

Es existieren verschiedene Verfahren zur Beschleunigung von DBSCAN, wovon wir hier nur die exakten Verfahren betrachten, da FM-DBSCAN ebenfalls exakt ist. GriDBSCAN [MM08] basiert auf einer Gitterzerlegung des $\mathbb{R}^d$. Auf den einzelnen Gitterzellen wird DBSCAN ausgeführt und die Ergebnisse anschließend gemischt. Durch die Zerlegung in kleinere Teilprobleme lässt sich eine entsprechende Beschleunigung erzielen. Die DBSCAN-Variante von Gan und Tao [GT15] ist ein gitterbasiertes Verfahren, bei dem zuerst die Objekte einer Zelle als Kern- oder Nicht-Kernobjekte markiert und anschließend die Zellen mittels einer dem Satz 1 ähnlichen Bedingung zu Clustern vermischt werden. Die Laufzeit beträgt $O((n \log n)^{4/3})$ für $d = 3$ und $O(n^{2-2/(\lceil d/2 \rceil + 1)+\delta})$ für $d \geq 4$, mit $\delta > 0$. G-DBSCAN [KR16] nutzt ebenfalls Leader-Umgebungen, nennt sie jedoch Gruppen. Die Erzeugung der Gruppen ist dem Verfahren von FM-DBSCAN sehr ähnlich. Die Gruppen werden verwendet, um die Berechnung der ϵ-Umgebung eines Objektes zu beschleunigen. Dabei bedienen sie sich einem Satz ähnlich dem Satz 3. Die Laufzeit beträgt $O(n^2)$. FM-DBSCAN hat diverse Vorteile gegenüber den hier aufgeführten Verfahren. Es werden keine zusätzlichen Parameter benötigt, anders als bei der Arbeit [MM08]. Zusätzlich ist FM-DBSCAN flexibel für beliebige Metriken einsetzbar. Die Arbeiten [GT15; MM08] sind auf den $\mathbb{R}^d$ beschränkt. Nur die Arbeit [KR16] ist ebenfalls metrisch. Zwar wird die Berechnung der ϵ-Umgebung beschleunigt, jedoch wird sie für jedes Objekt ausgeführt, was zu einem erhöhten Aufwand führt. FM-DBSCAN hingegen nutzt zielgerichteter die Erreichbarkeitsbegriffe von Objekten aus, um Beschleunigungen zu erzielen.

3 EFM-DBSCAN

In diesem Abschnitt wird Extended Fast Metric DBSCAN (EFM-DBSCAN) präsentiert, der basierend auf FM-DBSCAN, die folgenden neuen Konzepte einführt:

(a) Eine baumbasierte Partitionierung unter Nutzung der Excluded Middle Partitioning.

(b) Erweiterung der Objekte der Leader-Umgebungen um die Distanzen zu ihrem Leader.

Die bisherige FCFS-Partitionierung sucht für ein Objekt o linear über die aktuellen Leader-Umgebungen, um eine Leader-Umgebung $(l, N_\epsilon^L(l))$ zu finden, für die $d(o, l) \leq \epsilon$ gilt. Dabei ist sie von der Einfügereihenfolge abhängig. Ist diese ungünstig, führt das zu einer erhöhten Anzahl an Distanzberechnungen. Durch die Nutzung eine Baumes (a) können, anhand der räumlichen Lage, gezielt Teilbäume und somit Leader-Umgebungen ausgeschlossen werden, sodass man Distanzberechnungen und somit Rechenzeit einspart. Die in (b) durchgeführte Erweiterung ermöglicht schärfere Ausschlussbedingungen für die Sätze 2 und 3. Diese werden für eine effektivere Ausschlussstrategie anhand schrumpfender Leader-Umgebungen genutzt, um isCoreObject und isDirectDensityReachable zu beschleunigen.

3.1 Baumbasierte Partitionierung

In diesem Abschnitt wird die neue, baumbasierte Partitionierung zur Berechnung einer Leader-Partition vorgestellt. Zuerst wird das Prinzip der Excluded Middle Partitioning (EMP) vorgestellt, worauf die Partitionierung beruht. Anschließend erfolgt die Vorstellung des Excluded Middle Partitioning Tree (EMPT), der für die Partitionierung genutzt wird.

3.1.1 Excluded Middle Partitioning

Bei der EMP [Yi99] wird die Menge der Objekte O anhand eines gewählten Pivotobjektes $o_p \in O$ in die Mengen $S_1 = \{o \in O \mid d(o, o_p) \leq d_m - \rho\}$, $S_2 = \{o \in O \mid d_m - \rho < d(o, o_p) \leq d_m + \rho\}$ und $S_3 = \{o \in O \mid d(o, o_p) > d_m + \rho\}$ zerlegt. Bei d_m handelt es sich um den Median der Distanzen der Objekte $o \in O$ zu o_p. ρ ist ein vom Nutzer vorgegebener Parameter, der die Trenndistanz zwischen S_1 und S_3 festlegt. Für alle Objekte $o_1 \in S_1$, $o_2 \in S_3$ gilt $d(o_1, o_2) > 2\rho$. Eine solche beispielhafte Zerlegung ist in Abb. 1a zu sehen. Wesentlicher Vorteil der Unterteilung ist, dass für eine Bereichssuche des Objekts o_q in O bzgl. $\epsilon \leq \rho$ nur maximal in zwei der drei Mengen S_1, S_2 und S_3 gesucht werden muss. Um nun einen Baum mittels der EMP zu erhalten, müsste man S_1, S_2 und S_3 nur nach dem gleichen Prinzip erneut rekursiv weiter partitionieren. Die Pivotobjekte stellen die Knoten des Baumes dar.

(a) Excluded Middle Partitioning (b) Excluded Middle Partitioning Tree

Abb. 1: Beispiel für die Mengen S_1, S_2 und S_3.

3.1.2 Excluded Middle Partitioning Tree (EMPT)

Mit der EMP soll nun ein Baum konstruiert werden, der die Berechnung einer Leader-Partition beschleunigt. Hierzu muss der Baum folgende Anforderungen erfüllen. Es müssen zum einen die Leader-Umgebungen in den Baum abgebildet werden und zum anderen muss gewährleistet werden, dass zwei Leader eine Distanz größer als ϵ zueinander aufweisen.

Die Abbildung der Leader-Umgebungen erfolgt dadurch, dass anstatt der Objekte nun die Leader-Umgebungen die Knoten repräsentieren. Des Weiteren werden die Parameter der Excluded Middle Partitioning angepasst. Wir setzen $d_m = 3\epsilon$ und $\rho = \epsilon$ fest und beschränken die Objekte o von S_1 nach unten mit $\rho < d(o, l)$, sodass sich folgende drei Mengen ergeben: $S_1 = \{o \in O \mid \epsilon < d(o, l) \leq 2\epsilon\}$, $S_2 = \{o \in O \mid 2\epsilon < d(o, l) \leq 4\epsilon\}$ und $S_3 = \{o \in O \mid d(o, l) > 4\epsilon\}$. Alle Objekte o, die nicht in S_1, S_2 oder S_3 liegen, gehören zu der Leader-Umgebung des Knotens, mit l als Leader. Ein Beispiel für die neue Zerlegung mit einer Leader-Umgebung (innerer grauer Kreis) ist in Abb. 1b enthalten.

Um nun sicherzustellen, dass die Leader zweier Leader-Umgebungen eine Distanz größer ϵ haben, wird der Baum iterativ aufgebaut. Initialisiert wird der Baum mit der Leader-Umgebung $(o_s, \{o_s\})$ eines zufällig gewählten Objekts $o_s \in O$. Für jedes weitere Objekt o wird nun versucht, es in eine Leader-Umgebung $(l, N_\epsilon^L(l))$ im aktuellen Baum einzufügen, für die $d(o, l) \leq \epsilon$ gilt. Hierzu wird der Baum mittels einer Tiefensuche traversiert. Dazu wird die Distanz des Leaders l vom aktuellen Knoten zum Objekt o berechnet. Gilt $d(o, l) \leq \epsilon$, so wird o in $N_\epsilon^L(l)$ eingefügt und die Suche abgebrochen. Ansonsten wird entsprechend der Distanz $d(o, l)$ entweder im linken ($d(l, o) \leq 2\epsilon$), mittleren ($2\epsilon < d(l, o) \leq 4\epsilon$) oder rechten ($d(l, o) > 4\epsilon$) Teilbaum weitergesucht. Der linke, mittlere und rechte Teilbaum repräsentieren jeweils die Mengen S_1, S_2 und S_3 bezüglich des Leaders l. Sollte in diesen Teilbäumen ebenfalls keine Leader-Umgebung gefunden werden, so muss anschließend für den linken im mittleren, für den mittleren entweder im linken ($d(l, o) \leq 3\epsilon$) oder im rechten ($d(l, o) > 3\epsilon$) und für den rechten ggf. im mittleren ($d(l, o) \leq 5\epsilon$) Teilbaum weitergesucht werden. Die Suche in weiteren Teilbäumen muss durchgeführt werden, da die ϵ-Umgebung des Objektes o mehrere Teilbäume schneiden kann. Konnte keine Leader-Umgebung gefunden werden, so wird $(o, \{o\})$ als Blattknoten eingefügt. Hierfür wandert man den Baum von der Wurzel nach unten und steigt jeweils im aktuellen Knoten entweder in den linken, mittleren oder rechten Teilbaum ab. Wurden alle Objekte eingefügt, so bilden die Knoten unsere Leader-Partition. Den eben beschriebenen Algorithmus nennen wir EMPT-Partitionierung. Von Vorteil ist, dass keine zusätzlich Parameter benötigt werden. Ein Nachteil ist, dass der Baum nicht zwangsläufig balanciert ist, sodass er zu einer Liste entarten kann. Somit wird er nicht wesentlich schlechter als die FCFS-Partitionierung.

3.2 Erweiterung der Objekte der Leader-Umgebungen

Die Erweiterung der Objekte o einer Leader-Umgebung $(l, N_\epsilon^L(l))$ um die Distanzen $d(o, l)$ erfolgt durch die Abbildung $d^l : N_\epsilon^L(l) \to \mathbb{R}$, mit $d^l(o) = d(o, l)$. Die Distanzen werden nur einmal und nicht bei jedem Aufruf von d^l berechnet. Die erstmalige Berechnung erfolgt bei der Bestimmung der Leader-Partition, da dort die Distanzen zum Leader berechnet werden

müssen. Eine Aktualisierung von d^l erfolgt ggf. noch in der Neuverteilung der Objekte von FM-DBSCAN [Eg16]. Mit der Erweiterung werden nun schärfere Ausschlussbedingungen für die Sätze 2 und 3 formuliert. Diese werden dann dazu genutzt, um eine effektivere Ausschlussstrategie anhand schrumpfender Leader-Umgebungen zur Beschleunigung der Methoden isCoreObject und isDirectDensityReachable zu entwickeln.

3.2.1 Schärfere Ausschlussbedingungen

Sei d^l_{max} die maximale Distanz eines Objektes einer Leader-Umgebung zu seinem Leader, dann lassen sich die folgenden Sätze formulieren[3]:

Satz 4. Sei $q \in O$ ein Objekt und $(l, N^L_\epsilon(l))$ eine Leader-Umgebung in O. Ist $d(q,l) > \epsilon + d^l_{max}$, so gilt: $\forall o \in N^L_\epsilon(l) : d(o,q) > \epsilon$.

Satz 5. Seien $(l_1, N^L_\epsilon(l_1))$ und $(l_2, N^L_\epsilon(l_2))$ zwei Leader-Umgebungen in O. Ist $d(l_1,l_2) > \epsilon + d^{l_1}_{max} + d^{l_2}_{max}$, so gilt: $\forall o_1 \in N^L_\epsilon(l_1), o_2 \in N^L_\epsilon(l_2) : d(o_1,o_2) > \epsilon$.

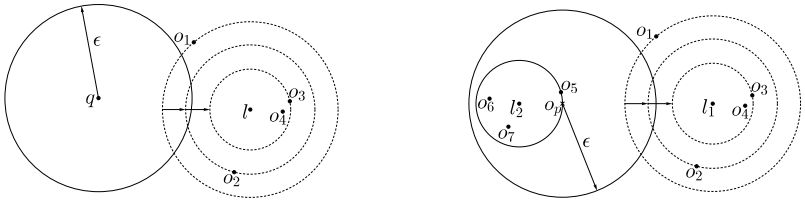
Die Sätze 4 und 5 sind eine Verbesserung der Sätze 2 und 3, da nicht mehr ϵ, sondern d^l_{max} als maximale Distanz für eine Leader-Umgebung verwendet wird. Da $d^l_{max} \leq \epsilon$ ist, kann mit den Sätzen 4 und 5 die Suche eher abgebrochen werden, als mit den Sätzen 2 und 3.

3.2.2 Ausschlussstrategie anhand schrumpfender Leader-Umgebungen

Eine verbesserte Ausschlussstrategie für isCoreObject und isDirectDensityReachable ergibt sich, wenn man den folgenden Zusammenhang betrachtet. Für eine Leader-Umgebung $(l, N^L_\epsilon(l))$ stellt das Tupel $(l_2, N^L_\epsilon(l_2))$, mit $l_2 = l$, $l_2 \in N^L_\epsilon(l_2)$ und $N^L_\epsilon(l_2) \subseteq N^L_\epsilon(l)$, ebenfalls eine Leader-Umgebung dar. Da $N^L_\epsilon(l_2)$ eine Teilmenge von $N^L_\epsilon(l)$ ist, gilt zusätzlich $d^l_{max} \geq d^{l_2}_{max}$. Diese Eigenschaft kann man sich zunutze machen, wenn man die Objekte o_i einer Leader-Umgebung $(l, N^L_\epsilon(l))$ immer absteigend sortiert nach $d^l(o_i)$ abarbeitet. Sei $o_1, o_2, \ldots, o_n$ eine solche Sortierung für $N^L_\epsilon(l)$, dann stellen die Tupel $(l_i, N^L_\epsilon(l_i))$, mit $l_i = l$, $N^L_\epsilon(l_i) = \{o_i, \ldots o_n\}$, $d^{l_i}_{max} = d(o_i, l)$ und $1 \leq i \leq n$, eine Folge von schrumpfenden Leader-Umgebungen dar, auf die man den Satz 4 für ein Objekt q anwenden kann. Analog kann man das für zwei schrumpfende Leader-Umgebungen und Satz 5 tun.

Beispiele für den Abbruch anhand von Satz 4 oder 5 sind in Abb. 2 zu sehen. Die Folge der schrumpfenden Leader-Umgebungen ist durch die gestrichelten Kreise und die schwarzen Pfeile dargestellt. In Abb. 2a kann bei der von o_3 begrenzten Leader-Umgebung anhand von Satz 4 abgebrochen werden, da es keine Überlappung mit der ϵ-Umgebung von q gibt. In Abb. 2b erfolgt ein Abbruch mit Satz 5 auch erst mit Objekt o_3. Die eingezeichnete ϵ-Umgebung gehört zu dem am nächsten liegenden Objekt o_p zu l_1, was noch in $(l_2, N^L_\epsilon(l_2))$ enthalten sein könnte. Solange die schrumpfenden Leader-Umgebungen von $(l_1, N^L_\epsilon(l_1))$ diese ϵ-Umgebung schneiden, ist ein Abbruch mit Satz 5 nicht möglich.

[3] Die Beweise sind aus Platzgründen in http://dbis.informatik.tu-cottbus.de/down/pdf/Leader_Umgebungen.pdf zu finden.



(a) Abbruch anhand von Satz 4 (b) Abbruch anhand von Satz 5

Abb. 2: Beispiele für schrumpfende Leader-Umgebungen und der Abbruch anhand von Satz 4 und 5

Für die Integration der Ausschlussstrategie werden beide Methoden angepasst. In isCoreObject muss in der Schleife, in der über die Objekte o_2 der aktuellen Leader-Umgebung $(l, N_\epsilon^L(l))$ iteriert wird, der Test von Satz 4 vor der Berechnung von $d(o, o_2)$ eingefügt werden, um die Suche für das zu Objekt o in $(l, N_\epsilon^L(l))$ frühzeitig abzubrechen.

Alg. 1 isDirectDensityReachable$((l_1, N_\epsilon^L(l_1)), (l_2, N_\epsilon^L(l_2)), L, \epsilon, minPts)$

```
1: if isCoreObject(l2, L, ϵ, minPts) ∨ |N_ϵ^L(l2)| = 1 then
2:     N2 := N_ϵ^L(l2); d_max^l2 = max_{o∈N2} d^l2(o);
3:     for all o1 ∈ N_ϵ^L(l1) do
4:         if d(l1, l2) > d^l1(o1) + d_max^l2 + ϵ then return false;
5:         if d(o1, l2) ≤ ϵ ∧ isCoreObject(o1, L, ϵ, minPts) = true then return true;
6:         for all o2 ∈ N2 do
7:             if d(o1, l2) > d^l2(o2) + ϵ then break;
8:             if d(o2, l1) ≤ ϵ ∧ (|N_ϵ^L(l2)| = 1∨ isCoreObject(o2, L, ϵ, minPts) = true) then return true;
9:             if d(o2, l1) > d^l1(o1) + ϵ then
10:                N2 := N2 \ {o2}; d_max^l2 = max_{o∈N2} d^l2(o);
11:                if N2 = ∅ then return false;
12:            if d(o1, o2) ≤ ϵ∧ isCoreObject(o1, L, ϵ, minPts) = true then
13:                if |N_ϵ^L(l2)| = 1∨ isCoreObject(o2, L, ϵ, minPts) = true then return true;
14: return false;
```

Die Anpassung von isDirectDensityReachable ist aufwendiger, weswegen der Code in Alg. 1 dargestellt ist. L ist die Menge der Leader-Umgebungen. Satz 4 wird für die Objekte aus $N_\epsilon^L(l_1)$ in der Z. 7 analog zu der Anpassung von isCoreObject überprüft. Um Satz 4 auch für die Objekte aus $N_\epsilon^L(l_2)$ anzuwenden, wird eine temporäre Kopie N_2 von $N_\epsilon^L(l_2)$ verwendet. Aus N_2 wird, falls Satz 4 anwendbar ist (Z. 9), das entsprechende Objekt entfernt und die neue maximale Distanz zum Leader l_2 ermittelt (Z. 10). Ist N_2 leer, kann die Methode als fehlgeschlagen abgebrochen werden (Z. 11). Der Test von Satz 5 befindet sich in der Z. 4. Der Test der direkten Dichte-Erreichbarkeit mittels Satz 1 ist in den Z. 1, 8 und 12–13 enthalten. Die sortierte Abarbeitung hat zusätzlich den Vorteil, dass gerade Objekte, die am Rand von Leader-Umgebungen liegen, auf die Distanz kleiner gleich ϵ getestet werden. Diese sind eher als die inneren Objekte dazu geeignet die direkte Dichte-Erreichbarkeit zu gewährleisten, sodass zeitiger abgebrochen werden kann. Nachteilig ist die Sortierung der Leader-Umgebungen. Dieser zusätzliche Aufwand fällt jedoch durch die eingesparten Distanzberechnungen nicht erheblich ins Gewicht (siehe Abschnitt 4).

4 Evaluation

EFM-DBSCAN wurde mit DBSCAN, G- und FM-DBSCAN verglichen. Alle Experimente liefen auf einem Intel® Core™ i7-2600K Prozessor. Die Daten wurden im Hauptspeicher gehalten. Als Effizienzmaß wurde die Anzahl der Distanzberechnungen (#Distanzen) und die Rechenzeit genutzt. Jeder Lauf wurde fünfmal mit einer zufällig generierten Permutation der Objekte ausgeführt und die Ergebnisse gemittelt. Als synthetische Datenkollektion kam *Gaussian Mixture Clusters* (12 Dimensionen)[4] zum Einsatz. Als reale Datenkollektion wurde *Caltech256* [GHP07] verwendet, die 30.607 Bilder enthält, für die das Feature *Dominant Color* (MPEG-7) extrahiert wurde. Für *Caltech256* wurde die Earth Mover's Distanz (EMD) und für *Gaussian Mixture Clusters* die euklidische Distanz genutzt.

Abb. 3 stellt die Skalierbarkeit der Verfahren mit wachsender Kollektionsgröße dar. EFM-DBSCAN benötigt deutlich weniger Distanzberechnungen und Rechenzeit als die anderen Verfahren. Gegenüber DBSCAN und G-DBSCAN konnte ein Faktor in der Größenordnung von 10 bis 10^2 eingespart werden. Im Vergleich zu FM-DBSCAN wird ein Faktor von 7 bis 17 weniger Distanzberechnungen und ein Faktor 3 bis 14 weniger Rechenzeit benötigt. Es zeigt sich also, dass die eingeführten Verbesserungen eine Beschleunigung erzielen und der zusätzliche Sortieraufwand vernachlässigbar ist. Obwohl G-DBSCAN gegenüber DBSCAN weniger Distanzberechnungen ausführt, ist die Rechenzeit höher. Dies ist auf den Mehraufwand zur Reduzierung der Distanzberechnungen und der kompletten Berechnung der ϵ-Umgebung zurückzuführen.

In Abb. 4 ist der Einfluss von ϵ auf die Performanz der Verfahren dargestellt. FM- und EFM-DBSCAN erzielen auf beiden Kollektionen die besten Ergebnisse. Wie erwartet, nähern sich die beiden Verfahren mit wachsendem ϵ immer weiter an. Das liegt an der sinkenden Anzahl Leader-Umgebungen, was das dazu führt, dass der Baum der EMPT-Partitionierung nicht ausreichend befüllt ist und die Häufigkeit der Tests der direkten Dichte-Erreichbarkeit sinkt. Somit werden die eingeführten Verbesserungen seltener angewendet. Dennoch erzielt EFM-DBSCAN gerade für aufwendig zu berechnende Metriken (EMD) und kleinere Werte für ϵ eine entsprechende Verbesserung gegenüber FM-DBSCAN. Für *Gaussian Mixture Clusters* und *Caltech256* wird bis zu einem Faktor 17 und 5 an Distanzberechnungen und bis zu einem Faktor 13 und 4 an Rechenzeit eingespart.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde das Verfahren EFM-DBSCAN vorgestellt, welches auf FM-DBSCAN basiert. Dabei wurden zwei neue Konzepte zu Reduzierung der Anzahl der Distanzberechnungen eingeführt: (a) eine baumbasierte Partitionierung und (b) die Erweiterung der Objekte einer Leader-Umgebung um die Distanzen zu ihrem Leader. Die Experimente zeigen, dass EFM- gegenüber FM-DBSCAN bis zu einem Faktor 17 weniger Distanzberechnungen und bis zu einem Faktor 13 weniger Rechenzeit benötigt. Im Vergleich zu DBSCAN wurde bis zu einem Faktor 10^4 eingespart. In zukünftigen Arbeiten soll die Wahl von ϵ und *minPts* bzgl. einer guten Qualität und hohen Effizienz untersucht werden.

[4] Generiert mittels RapidMiner (https://rapidminer.com/) in den Größen n = 10.000, 20.000, . . . , 100.000.

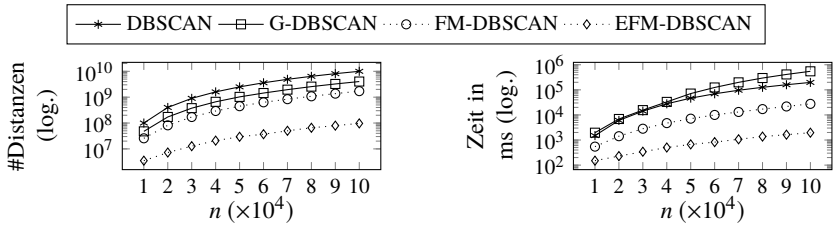
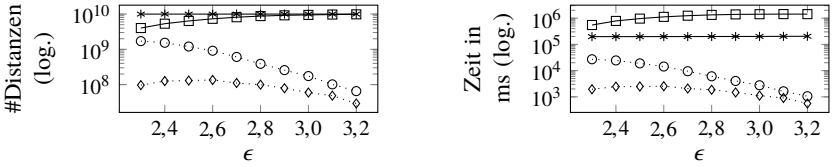
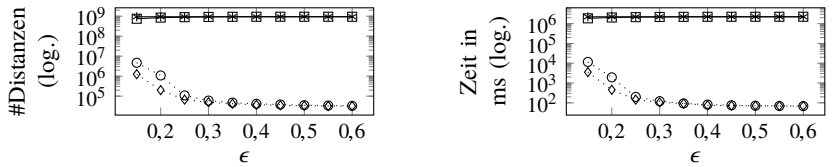


Abb. 3: Skalierbarkeit mit der Kollektionsgröße n auf *Gaussian Mixture Clusters* ($minPts = 5$, $\epsilon = 2{,}3$)



(a) *Gaussian Mixture Clusters* ($minPts = 5$, $n = 100.000$)



(b) *Caltech256* ($minPts = 2$, $n = 30.607$)

Abb. 4: Performanz für variierendes ϵ

Literatur

[Eg16] Egert, P.: FM-DBSCAN: Ein effizienter, dichte-basierter Clustering-Algorithmus. In: Proc. of the 28th GI-Workshop GvDB. S. 44–49, 2016.

[Es96] Ester, M.; Kriegel, H.; Sander, J.; Xu, X.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: Proc. of the 2nd Int. Conf. on Knowledge Discovery and Data Mining. S. 226–231, 1996.

[GHP07] Griffin, G.; Holub, A.; Perona, P.: Caltech-256 Object Category Dataset, Techn. Ber. 7694, California Institute of Technology, 2007.

[GT15] Gan, J.; Tao, Y.: DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In: Proc. of the 2015 ACM SIGMOD. S. 519–530, 2015.

[KR16] Kumar, K. M.; Reddy, A. R. M.: A fast DBSCAN clustering algorithm by accelerating neighbor searching using Groups method. Pattern Recognition 58/, S. 39–48, 2016.

[MM08] Mahran, S.; Mahar, K.: Using grid for accelerating density-based clustering. In: Proc. of 8th IEEE Int. Conf. on Computer and Info. Techn. S. 35–40, 2008.

[Yi99] Yianilos, P. N.: Excluded middle vantage point forests for nearest neighbor search. In: In DIMACS Implementation Challenge, ALENEX'99. 1999.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 237

Detection and Implicit Classification of Outliers via Different Feature Sets in Polygonal Chains

Michael Singhof[1], Gerhard Klassen[2], Daniel Braun[1], Stefan Conrad[1]

Abstract: Many outlier detection tasks involve a classification of outliers of different types. Most standard procedures solve this problem in two steps: First, an outlier detection algorithm is carried out, which is normally trained on outlier free data, only, since the samples of outliers are limited. Second, the outliers detected in that step, are classified with a conventional classification algorithm, that needs samples for all classes. However, often the quality of the classification is lowered due to the small number of available samples.

Therefore, in this work, we introduce an outlier detection and classification algorithm, that does not depend on training data for the classification process. Instead, we assume, that different kinds of outliers are inferred by different processes and as such should be detected by different outlier detection approaches. This work focuses on the example of outliers in mountain silhouettes.

Keywords: Anomaly & Outlier Detection, Classification, Image Segmentation

1 Introduction

The detection of outliers in a data set occurs in many contexts, ranging from clustering algorithms such as DBSCAN [Es96] to credit card fraud detection [CS98, Ch99], function tests of aircraft engines [Ab16], or the detection of cyber attacks [La04, CBG12] and many other application areas. In some cases, like clustering, it is sufficient to just erase any point that is either noise or an anomaly. In other cases, like function tests, the outliers are of primary interest. A general problem in those cases is the fact, that normally, none or very few outlier instances are known. Therefore, outlier detection is mostly treated as a single class problem, where for each data point it is rated, whether that point is normal or not. If it is not judged as being normal, it is treated as an outlier.

In some cases, however, a further classification of outliers is necessary. In [BSC16], we presented a system that is able to find a mountain's silhouette in a photo. It utilises an outlier detection algorithm to get rid of errors during the segmentation step, that belong to different classes. In this case, it is important to differentiate between these classes, because errors of different kinds get corrected in different ways.

The remainder of this paper is structured as follows: Chapter 2 explains and motivates the problem we try to solve, chapter 3 gives an overview of related work. In chapter 4 we describe

[1] Heinrich-Heine-Universität Düsseldorf, Institut für Informatik, Universitätsstraße 1, 40225 Düsseldorf, {singhof, braun,conrad}@cs.uni-duesseldorf.de
[2] Heinrich-Heine-Universität Düsseldorf, Institut für Informatik, Universitätsstraße 1, 40225 Düsseldorf, gerhard.klassen@uni-duesseldorf.de

our approach to outlier detection and how it can be used for an implicit classification. We evaluate our approach in chapter 5 and finally draw a conclusion in chapter 6.

2 Motivation and Problem Description

In this work, we present an approach to outlier detection and classification on polygonal chains, that are given by an image segmentation algorithm. The aim of this framework as a whole is the automatic annotation of mountain photos, by detecting the silhouette of a mountain in a given image and then comparing it to a set of reference silhouettes.

During the segmentation step several problems can occur that might obscure the silhouette: First, there can be obstacles in the photo that are in front of the mountain, such as trees, buildings or persons. In order to extract an exact silhouette it is necessary to take note of such obstacles and ignore them in the silhouette matching step. Second, segmentation errors can occur that can by caused by a low contrast between sky and foreground. This happens if clouds occur close or overlapping to the silhouette, if snowfields appear next to light sections of the sky or for other reasons where contrast between sky and foreground is minor. Some of these errors are shown in figure 1, that has obstacles in the form of trees on the left side and segmentation errors due to low contrast on the right hand side.

Fig. 1: Examples for different errors in a mountain silhouette.

Since we use an adaptive segmentation algorithm, it is possible to correct errors if we detect them. Figure 2 gives an overview of the architecture of the segmentation module. In general, a grid is laid over the image. For the initial segmentation, the same parameters are used for every cell, although these can be changed during the adaptive process. Then, the segmentation algorithm computes a silhouette from the image, which is passed to the outlier detection step. When no outliers are found, the silhouette is inferred as clean and the algorithm terminates. In the case that outliers are found, these are passed to the classification module. If an outlier gets classified as being an obstacle, it is removed by replacing it by a straight line. Otherwise, if an outlier is classified as a segmentation error, it is passed to the segmentation module. There, the parameters for the affected grid cells are changed to better accommodate the local circumstances.

This work concentrates on the detection and classification of outliers. A general problem with outlier detection, as mentioned in the introduction, is the fact, that in most cases no exhaustive collection of examples for all shapes of outliers exists. This is the case with

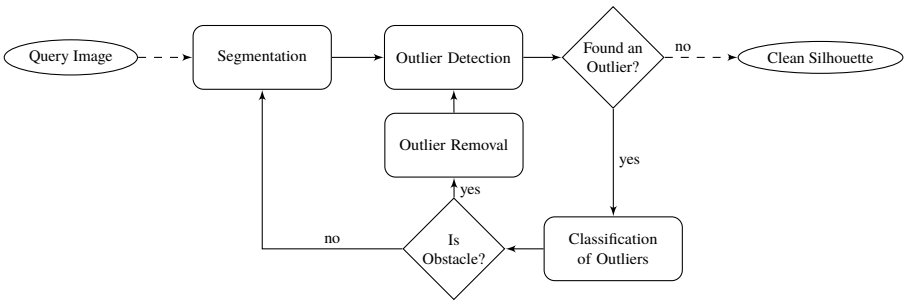


Fig. 2: Flow diagram of adaptive image segmentation.

our problem, too. Therefore, our outlier detection algorithm is trained on normal data, only. However, for a regular classification samples of all classes are needed. One can argue here, that for most outliers that are found, a small collection of examples is sufficient to differentiate between the given classes. In previous work [BSC16], we have used such a solution to acceptable results.

In our current approach, that has been outlined in [SBC16], we only use the geometric properties of the silhouette to depict outliers. In this work, however, we want to implicitly classify the detected outliers by using different sets of features for the different types of outliers. Our notion is, that those different outliers are produced by different mechanisms and thus can be detected by looking at different attributes: On one hand, segmentation errors occur in regions with unusual low contrast along the silhouette. On the other hand, obstacles usually have borders to the sky that are as sharp as the rest of the silhouette in the surroundings of the obstacle but have unusual forms for mountain silhouettes.

3 Related Work

There is an abundance of work on outlier detection, beginning with statistical models [Ha80, BL94] and general definitions of outliers [BC83] to very specialised applications of outlier detection. Some of them have been already mentioned in the introduction such as [CS98, La04, CBG12, Ab16]. Since, to our knowledge, there is no work on outlier detection on polygonal chains in general, the field that is closest related to ours is outlier detection on time series. There are two major problems for outliers in time series, namely the finding of change points and the finding of unusual subseries. A change point is a certain point in time, where the time series changes its behaviour drastically. This has, among others, researched in [FP99, KS09]. Well known approaches to the problem of finding unusual parts of time series include HotSax[KLF05] and specialisations of it such as [PLD10, BA11b, KA12].

The basic idea of HotSax is finding the strongest discord. A discord is a subsequence of a time series, that does not fit the general shape of the time series it lies in. HotSax computes the one most unusual part of a time series consisting of k points where k is given by the user. In contrast to this, for a given polygonal chain, the target of this work is to find all

outliers of arbitrary lengths for a given polygonal chain, including the possibility of not finding an outlier at all if there is none in the data.

The target of mountain recognition was first tackled in [Ba12]. Together with this paper, a corpus of 203 annotated images was released, which is often used in this field. In contrast to our approach, the approach by Baatz et al. requires human intervention in some cases. Kim et al. [Ki11] introduced a skyline detection algorithm that uses a Canny edge detection [Ca86],first, and then filters the resulting edges in order to get silhouette edges, only. One disadvantage of this algorithm is the fact, that it does not find a continuous silhouette but in most cases only parts of it. The authors of [Ah15] use a combination of different techniques, both edge-less and edge-based in order to come up with a skyline. Baboud et al. [Ba11a] use a similar technique in order to annotate mountains, that relies on GPS coordinates and does not extract a skyline or silhouette in particular.

4 Outlier Detection and Classification

As mentioned in section 2, the segmentation part of our framework passes a silhouette in the form of a polygonal chain to the outlier detection part. This gets searched for untypical parts and then those parts are given to the classification module. In this section, we describe the current approach two-dimensional approach and introduce changes to the work presented in [BSC16, SBC16], namely the addition of further dimensions and the proposal of merging strategies.

To understand the nature of outliers, we first have to introduce the construct of the silhouette. Visually speaking, the silhouette is the border in the image, that separates the sky from the foreground, or in our case, that separates the mountain from everything above the mountain, that might include objects in front of the mountain.

Formally, we define a silhouette as follows:

Definition 1. Let $S = (p_1, \ldots, p_n)$ be a sequence of points $p_i = (x_i, y_i) \in [0, x_{\max}] \times [0, y_{\max}]$ for an image of the size of $x_{\max} \times y_{\max}$ pixels. S is called a *silhouette* if the points p_1 and p_n lie on the borders of the image.

Given a silhouette S, an outlier is a sub-sequences $O = (p_i, \ldots, p_j)$, $1 \leq i < j \leq n$, that marks an unusual part of the silhouette. This unusualness is expressed by an anomaly score on single points of a silhouette. The bigger that anomaly score is, the more unusual a point is. An outlier consists of a series of points that each have a high anomaly score.

4.1 Outlier Detection

After the previous section introduced a general idea of the term outlier, in this section we describe how outliers are computed. Figure 3 gives an overview over the outlier detection process.

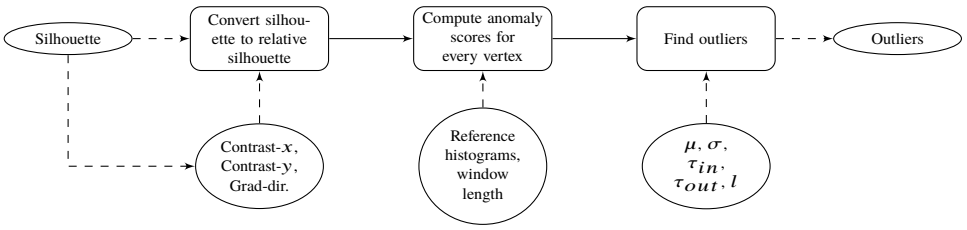


Fig. 3: Flow diagram of outlier detection.

It can be seen there, that the process is essentially split in three steps; the first of which is the conversion of a silhouette S to a so-called relative silhouette RS. In contrast to a silhouette, in a relative silhouette we give the coordinates of every vertex relatively to the coordinates of its predecessor in the silhouette. In our case we chose a polar coordinate like notation. We also add additional information to the silhouette, namely contrast values in x and y direction, as well the gradient direction:

Definition 2. Let $S = (p_1, \ldots, p_n)$ be a silhouette. $RS = (v_1, \ldots, v_n)$ is called the *relative silhouette of* S if $v_1 = (0, 0, cx_1, cy_1, g_1)$ and for each $v_i = (sl_i, a_i, cx_i, cy_i, g_i)$ for $i \in [2, n]$, $sl_i = |p_i - p_{i-1}|$ denotes the length of the the line segment between p_i and p_{i-1}, a_i denotes the angle between that line segment and the x-axis, cx_i and cy_i give the contrast at the point p_i in direction of the x, respectively y axis, and g_i denotes the contrast direction at point p_i.

The computation of the relative silhouette for a given silhouette is straight forward and can be carried out in $O(n)$, for n the number of points in the silhouette S. This is done in order to ensure, that similar structures in the silhouette that lie in different parts of the image are represented in a similar fashion, without the need to do any further computations.

The second step in the process is the computation of the anomaly scores of the single vertices of the relative silhouette RS. As can be seen in figure 3, in this step, additional information is needed, namely reference histograms and a parameter called window length. A relative silhouette – or a part of one – can be easily transformed into a histogram. The histogram's bins consist of up to five dimensions, depending on the chosen features. The reference histograms are histograms derived from error free silhouettes. More on their computation can be found in [SBC16] for the two dimensional case. Computation for other numbers of dimensions are analogous. Let us assume that we have k reference histograms $H^1_{ref}, \ldots, H^k_{ref}$. Then we use a sliding window with the size given by the parameter window size. For every window we compute the corresponding histogram H and with that the distance to the reference histograms as $d = \min_{1 \leq i \leq k} \text{dist}(H, H^i_{ref})$, for any histogram distance function dist.

We store d for every vertex that has been part of the sliding window so that every vertex v_i gets a collection D_i of distance values.

Definition 3. Let D_i be the collection of distance values for a point v_i. Then we call $an(v_i) = \frac{1}{|D_i|} \sum_{d \in D_i} d$ the *anomaly score* of v_i.

As with the silhouette conversion, it is obvious, that the anomaly scores can be computed in $O(n)$, since the number of bins of the histograms and therefore the histogram distance function are independent of the number of points in the silhouette.

The third and last step is the actual outlier detection. Again, as shown in figure 3, some additional parameters are necessary. Of these, μ is the mean of the anomaly score distribution computed on the reference data and σ is the standard deviation. These are computed together with the reference histograms from the previous step. In contrast to this, τ_{in}, τ_{out} and l are parameters given by the user. Here, τ_{in} and τ_{out} are thresholds for the anomaly scores and l is the minimal number of points that an outlier has to consist of.

Definition 4. Let $RS = (v_1, \ldots, v_n)$ be the relative silhouette of an image with corresponding anomaly scores $an(v_i)$ for vertex v_i, reference anomaly score distribution mean μ and standard deviation σ and two thresholds $0 < \tau_{out} < \tau_{in}$.

Then we call v_i a *weak anomaly* if $an(v_i) \geq \mu + \tau_{out} \cdot \sigma$ and a *strong anomaly* if $an(v_i) \geq \mu + \tau_{in} \cdot \sigma$.

We use a double threshold technique since our observations show that on one hand, if we chose only one relatively high threshold, the detected outliers would often be too small. On the other hand, if we chose a single low threshold, we would find many false positives. Therefore, due to the double thresholds, we can restrain the number of detected outliers by τ_{in} but are able to expand those outlier by choosing a lower value for τ_{out}.

We can now define an outlier in our context as given in [BSC16]:

Definition 5. Let $l > 0$, and $RS = (v_1, \ldots, v_n)$ be a relative silhouette. We call $o = (v_i, \ldots, v_j)$ an *l-outlier* if the following is true:

1. For all v_k, $i \leq k \leq j$, it holds that v_k is a weak anomaly.
2. There exist $m_1, m_2 \in \{i, \ldots, j\}$ such that $m_2 - m_1 \geq l$ and for all v_k, $m_1 \leq k \leq m_2$, it holds that v_k is a strong anomaly.

An outlier $o = (v_i, \ldots, v_j)$ is called a *maximum l-outlier* if and only if neither $(v_{i-1}, \ldots, v_j)$ nor $(v_i, \ldots, v_{j+1})$ are l-outliers.

Our goal is to find all maximum outliers in the silhouette. This is done by iterating over all vertices in the silhouette and then react as noted in table 1. There, s_{out} is a variable that stores the position of a possible start of the outer part of an outlier, or nf if no start has been found yet, s_{in} stores the possible start of an inner part of an outlier, and e_{in} stores the end of an inner outlier. The minimum inner length is given by l and v_i denotes the current vertex.

This can be done in linear time as well, so that the whole outlier detection algorithm can be executed in $O(n)$.

Status of variables			Status of current vertex v_i		
s_{out}	s_{in}	e_{in}	no anomaly	weak anomaly	strong anomaly
nf	nf	nf	—	set $s_{out} = i$	set $s_{in} = s_{out} = i$
found	nf	nf	set $s_{out} = nf$	—	set $s_{in} = i$
found	$i - s_{in} \leq l$	nf	set $s_{out} = s_{in} = nf$	set $s_{in} = nf$	—
found	$i - s_{in} > l$	nf	set $o_{in} = i$, save o	set $e_{in} = i$	—
found	$i - s_{in} > l$	found	save o	—	—

Tab. 1: Decision matrix for outlier detection.

Number of features	Used features	Outlier type
2	Length, Angle	Obstacle
3	Contrast x, Contrast y, Gradient dir.	Segmentation
5	Length, Angle, Contrast x, Contrast y, Gradient dir.	Segmentation

Tab. 2: Used feature set and their implication on outlier types.

4.2 Merging and Classification of Outliers

If only one feature set is used for the outlier detection, overlapping outliers do not occur, since an outlier is expanded first. That means, we enlarge an outlier, first, before we proceed with searching for the next outlier in the next part of the silhouette. However, if we carry out the outlier detection for different feature sets, in order to use an implicit classification, it is possible for outliers to overlap. We then have to decide, how to deal with those outliers and which class they should have.

The basic assumption for the classification is, that the feature set in which an outlier occurs, indicates the type of outlier. As described in section 2, outliers in the contrast features hint at segmentation errors while outliers with normal contrast but unusual shape hint at obstacles. Table 2 gives an overview of the feature sets we use and what they mean for the outlier types.

We have developed three strategies for the merging of outliers. The first is called “Merge” and, for a set of overlapping outliers creates an outlier that begins at the lowest start index and ends at the highest end index. The type is then computed as the type with the longest inner outlier, i.e. the part of the outlier, that consists of strong anomalies, of the involved outliers. The second strategy, “Merge to Segmentation”, depicts the merged outlier as being a segmentation error if at least one outlier from a segmentation error feature set is involved. Finally, “Split and Merge” is the most complex strategy. Here we look, if the inner parts of outliers intersect. If this is the case, we merge the outliers with the “Merge” strategy. Otherwise, we split the combined outlier at the middle between the inner parts, if both outer outliers are reaching over the middle. Otherwise, the split is performed as close to the middle as possible. The single parts of the outlier then get merged and classified by the “Merge” strategy.

5 Evaluation

The evaluation is carried out on a data set, which consists of 3580 outlier vertices forming 114 outliers that have been manually marked in 14 silhouettes. The silhouettes are automatically detected by the segmentation part of AdaMS, that is described in [BSC16], although without using the adaptive improvement. This is in order to ensure typical outliers for our application scenario. All outlier detection algorithm variants have been trained on 48 mostly outlier free silhouettes. Clustering has been carried out 1000 times and the clustering with the lowest quadratic distances has been chosen, and the number of clusters and reference histograms k has been set to 30.

Method	Found outliers of length						Prec.	Recall	F_1
	$(0,5]$	$(5,10]$	$(10,20]$	$(10,50]$	>50	Total			
2d	6	21	12	19	14	72	70	70	70
3d	4	11	11	10	14	50	57	66	61
5d	15	36	22	21	15	109	52	81	66
Combined3	8	26	19	19	15	87	58	88	73
Combined5	15	38	23	21	15	112	51	88	70

Tab. 3: Comparison of single and combined approaches.

Table 3 shows the detection results for the single feature sets and the feasible combinations. It can be seen here, that the two dimensional method yields the best results in respect to precision and F_1 measure, while the five dimensional approach finds the most outliers and has the highest recall of the single feature sets. The three dimensional approach has the worst results, however this is as expected, since it is only able to detect outliers that have very unusual contrast values, i.e. segmentation errors. The five dimensional approach, is best suited to find most, since it is the only approach that uses all kinds of features by itself.

Combined3 gives the results for the combination of the two and three dimensional feature sets. It can be seen here, that the number of hit outliers rises significantly in comparison to the single methods, so that this combination is feasible. Recall reaches 88%, while precision is at 58%, resulting in the best F_1 value of all tested approaches. As expected, the combination of the two and the five dimensional feature set is not as good in respect to precision, since for the five dimensional feature set on its own, precision is already rather low. On the other hand, the number of detected outliers, especially shorter ones is nearly complete, as nearly all outliers have been found.

Strategy	Right	Wrong	Both
Merge	51	25	0
Merge to Seg.	51	25	0
Split and Merge	43	31	3

(a) Two features and three features.

Strategy	Right	Wrong	Both
Merge	69	31	1
Merge to Seg.	75	24	0
Split and Merge	68	41	11

(b) Two features and five features.

Tab. 4: Classification results for combined approaches.

Tables 4a and 4b show the classification results for the hit outliers for the Combined3 and Combined5 approach, respectively. Since the detection is not always exact, we declare a

detected outlier as classified correctly, if it intersects with an manually marked outlier of the same type, and we declare it incorrect if it does intersect with an marked outlier of the other type.

The results show, that, without any training data, there are up to 75% right classifications with the "Merge to Segmentation" merging strategy and better results for the combination with the five features approach. The lower classification rate for Combined3 seems to be due to the fact, that in some cases, the detected silhouette around obstacles is not entirely exact. As can be seen in figure 1, at some points the silhouette is inside the sky. So technically there is a slight segmentation error, that lowers the contrast, over the obstacle. We expect, that in the full adaptive context, in a first step this would be fixed and afterwards, the real obstacle would be detected as such.

6 Conclusion

In this work, we have argued, that in cases, where an outlier detection problem and an outlier classification problem, are tackled, it might be more feasible to instead regard the problem as multiple outlier detection problems and carry out the classification implicitly by the outlier detection algorithm, that detects a given outlier. For the example of outliers in silhouettes of segmentations of mountain images, we have shown, that not only such a classification is possible, but that the usage of more than one outlier detection variant even increases the total number of detected outliers.

We believe, that the basic idea of our approach, namely the usage of separate outlier detection methods and an implicit classification based on those, is adaptable to a other outlier detection problems. Future work therefore will focus on testing such frameworks on other problems.

References

[Ab16] Abdel-Sayed, Mina; Duclos, Daniel; Faÿ, Gilles; Lacaille, Jérôme; Mougeot, Mathilde: Dictionary Comparison for Anomaly Detection on Aircraft Engine Spectrograms. In: Proc. of the MLDM 2016. 2016.

[Ah15] Ahmad, Touqeer; Bebis, George; Nicolescu, Monica; Nefian, Ara; Fong, Terry: Fusion of Edge-less and Edge-based Approaches for Horizon Line Detection. In: 6th IEEE International Conference on Information, Intelligence, Systems and Applications (IISA'15), Corfu, Greece, July 6-8, 2015. IEEE, 2015.

[Ba11a] Baboud, Lionel; Čadík, Martin; Eisemann, Elmar; Seidel, Hans-Peter: Automatic Photo-to-terrain Alignment for the Annotation of Mountain Pictures. In: Proc. of the 2011 IEEE Conference on Computer Vision and Pattern Recognition. 2011.

[BA11b] Buu, Huynh Tran Quoc; Anh, Duong Tuan: Time Series Discord Discovery Based on iSAX Symbolic Representation. In: Third International Conference on Knowledge and Systems Engineering. 2011.

[Ba12] Baatz, Georges; Saurer, Olivier; Köser, Kevin; Pollefeys, Marc: Large Scale Visual Geo-Localization of Images in Mountainous Terrain. In: Computer Vision - ECCV. 2012.

[BC83] Beckman, Richard J; Cook, R Dennis: Outlier..... s. Technometrics, 25(2), 1983.

[BL94] Barnett, Vic; Lewis, Toby: Outliers in Statistical Data. 1994.

[BSC16] Braun, Daniel; Singhof, Michael; Conrad, Stefan: AdaMS: Adaptive Mountain Silhouette Extraction from Images. In: Proc. of the MLDM 2016. 2016.

[Ca86] Canny, John: A Computational Approach to Edge Detection. Pattern Analysis and Machine Intelligence, IEEE Transactions on, PAMI-8(6), Nov 1986.

[CBG12] Catania, Carlos A; Bromberg, Facundo; Garino, Carlos García: An Autonomous Labeling Approach to Support Vector Machines Algorithms for Network Traffic Anomaly Detection. Expert Systems with Applications, 39(2), 2012.

[Ch99] Chan, Philip K; Fan, Wei; Prodromidis, Andreas L; Stolfo, Salvatore J: Distributed Data Mining in Credit Card Fraud Detection. IEEE Intelligent Systems and Their Applications, 14(6), 1999.

[CS98] Chan, Philip K; Stolfo, Salvatore J: Toward Scalable Learning with Non-Uniform Class and Cost Distributions: A Case Study in Credit Card Fraud Detection. In: KDD. volume 98, 1998.

[Es96] Ester, Martin; Kriegel, Hans-Peter; Sander, Jörg; Xu, Xiaowei: A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: Proc. of the KDD 1996. 1996.

[FP99] Fawcett, Tom; Provost, Foster: Activity Monitoring: Noticing Interesting Changes in Behavior. In: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining. 1999.

[Ha80] Hawkins, Douglas M: Identification of Outliers. 1980.

[KA12] Khanh, Nguyen Dang Kim; Anh, Duong Tuan: Time Series Discord Discovery Using WAT Algorithm and iSAX Representation. In: Proceedings of the Third Symposium on Information and Communication Technology. 2012.

[Ki11] Kim, Byung-Ju; Shin, Jong-Jin; Nam, Hwa-Jin; Kim, Jin-Soo: Skyline Extraction Using a Multistage Edge Filtering. World Academy of Science, Engineering and Technology, 55, 2011.

[KLF05] Keogh, Eamonn; Lin, Jessica; Fu, Ada: Hot Sax: Efficiently Finding the Most Unusual Time Series Subsequence. In: Fifth IEEE International Conference on Data Mining (ICDM'05). 2005.

[KS09] Kawahara, Yoshinobu; Sugiyama, Masashi: Change-Point Detection in Time-Series Data by Direct Density-Ratio Estimation. In: Proc. of 2009 SIAM International Conference on Data Mining (SDM2009),. 2009.

[La04] Laskov, Pavel; Schäfer, Christin; Kotenko, Igor; Müller, K-R: Intrusion Detection in Unlabeled Data with Quarter-sphere Support Vector Machines. Praxis der Informationsverarbeitung und Kommunikation, 27(4), 2004.

[PLD10] Pham, Ninh D; Le, Quang Loc; Dang, Tran Khanh: HOT aSAX: A Novel Adaptive Symbolic Representation for Time Series Discords Discovery. In: Asian Conference on Intelligent Information and Database Systems. 2010.

[SBC16] Singhof, Michael; Braun, Daniel; Conrad, Stefan: Finding Trees in Mountains – Outlier Detection on Polygonal Chains. In: Proc. of the Conference LWDA 2016. 2016.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 247

Efficient Batched Distance and Centrality Computation in Unweighted and Weighted Graphs

Manuel Then[1] Stephan Günnemann[2] Alfons Kemper[3] Thomas Neumann[4]

Abstract: Distance and centrality computations are important building blocks for modern graph databases as well as for dedicated graph analytics systems. Two commonly used centrality metrics are the compute-intense closeness and betweenness centralities, which require numerous expensive shortest distance calculations. We propose batched algorithm execution to run multiple distance and centrality computations at the same time and let them share common graph and data accesses. Batched execution amortizes the high cost of random memory accesses and presents new vectorization potential on modern CPUs and compute accelerators. We show how batched algorithm execution can be leveraged to significantly improve the performance of distance, closeness, and betweenness centrality calculations on unweighted and weighted graphs. Our evaluation demonstrates that batched execution can improve the runtime of these common metrics by over an order of magnitude.

Keywords: Graph Databases, Graph Analytics, Closeness Centrality, Betweenness Centrality

1 Introduction

Recently, there has been growing interest in graph analytics as a means of analyzing large social networks, web graphs, road networks and gene interaction graphs. This lead to the creation of graph databases and dedicated graph analytics systems like Pregel [Ma10] or PGX [Ho15]. A common graph analytics use case is ranking vertices by importance to find the most important vertices. This can, for example, be used to find the most influential users in a social network, and, thus, to improve the effectiveness of targeted advertising campaigns. Algorithms that determine the importance of vertices are called *centrality algorithms*.

In practice, different centrality algorithms are used. They cover specific use cases and cannot be used interchangeably [NSJ11]. Simple degree-based centrality algorithms like degree centrality [Fr78] directly use the vertices' number of incident edges as a measure for their importance, and can be used as a first indication of relevant vertices. The PageRank [BP98] algorithm extends this idea by iteratively propagating vertices' influence through the graph. Both degree centrality and PageRank are well-researched and can be efficiently computed for large graphs. Other popular centrality algorithms are based on the notion of paths. The *closeness centrality* [Fr78] metric ranks vertices by their average geodesic distance to all other vertices, i.e., the length of the shortest paths to these other vertices. A vertex is

[1] TU Munich, Department of Informatics, Boltzmannstraße 3, 85748 Garching, then@in.tum.de
[2] TU Munich, Department of Informatics & Institute for Advanced Study, Boltzmannstraße 3, 85748 Garching, guennemann@in.tum.de
[3] TU Munich, Department of Informatics, Boltzmannstraße 3, 85748 Garching, kemper@in.tum.de
[4] TU Munich, Department of Informatics, Boltzmannstraße 3, 85748 Garching, neumann@in.tum.de

considered more central when it can reach other vertices in fewer steps; closeness centrality, hence, measures how fast information can propagate from a vertex through the network. In contrast, *betweenness centrality* [Fr78] counts the number of shortest paths between any two vertices that a vertex v is on. Thus, it is a measure for how much communication goes through v, and, as a result, for how much v can influence communication [NSJ11]. Closeness and betweenness centrality are computationally very expensive metrics with their complexities of $O(|V|^2 + |V| * |E|)$ and $O(|V| * |E|)$, respectively, on unweighted graphs, and $O(|V|^2 * |E|)$ and $O(|V| * |E| + |V|^2 * \log |V|)$, respectively, on weighted graphs [Br01]. The algorithms' high complexity on weighted graphs is caused by the required all pairs geodesic distance computations. This makes computing exact closeness and betweenness centralities prohibitively expensive on large-scale graphs that are used in practice. Hence, these metrics are often only approximated by means of sampling [EW01, Ba07].

To significantly reduce the runtimes of exact closeness and betweenness centrality computation, and to improve the accuracy of approximate centrality computation within a given time frame, we propose batched algorithm execution. In *batched algorithm execution* multiple executions of the same algorithm from different source vertices are run concurrently. They are synchronized to share common graph element accesses. Thus, batched algorithm execution amortizes the costs of random data accesses that are inherent to graph analytics. While this allows to greatly reduce the overall runtime of graph analytics algorithms, batched execution introduces new tradeoffs and makes necessary novel data structures.

In this paper we propose batched algorithms that efficiently calculate distances and centrality metrics in unweighted and weighted graphs. To that end, we revisit the existing multi-source breadth-first search algorithm MS-BFS [Th14] for batched closeness centrality in unweighted graphs and improve its efficiency by means of a constant-time batch counter. Moreover, we propose a batched betweenness centrality algorithm for unweighted graphs that extends MS-BFS to allow reverse traversal and low-overhead vertex predecessor detection. We then further the principles of batched multi-source execution and show how they can be applied to geodesic distance, closeness centrality and betweenness centrality computation in weighted graphs.

Our contributions are as follows.

- We introduce batched algorithm execution and explain how an algorithm can be run concurrently from multiple source vertices and share common data accesses.
- We present batched algorithms for closeness centrality and betweenness centrality for unweighted graphs.
- We propose batched geodesic distance, closeness centrality and betweenness centrality algorithms for weighted graphs.
- We evaluate our algorithms using multiple synthetic and real-world datasets.

The paper is structured as follows. After this introduction, in Section 2 we give a short overview of the terminology used in this paper. In Section 3 we present the concept of batched algorithm execution, which we apply to distance and centrality algorithms on unweighted

and weighted graphs in Sections 4 and 5, respectively. We evaluate our algorithms in Section 6. Section 7 elaborates on related work and Section 8 gives our conclusions.

2 Background

In this section we introduce the terminology and algorithms used throughout this paper. Furthermore, we explain the MS-BFS algorithm which we use as the basis of batched algorithm execution.

We define a graph G as the tuple (V, E) of vertices V and edges $E \subseteq V \times V$. Further, vertices and edges may have an arbitrary number of named properties attached to them.

2.1 Geodesic Distance

The *geodesic distance* between two vertices u and v is the length of the shortest path from u to v. In this paper we only consider the single-source shortest path case. For unweighted graphs, geodesic distance is measured in the number of traversed edges between u and v, which can efficiently be derived using a breadth-first search (BFS) from u. In weighted graphs, the distance from u to v is the sum of edge weights along the path between them. Depending on the type of graph and its topology, various algorithms exist to compute the geodesic distances. In this paper, we evaluate batched variants of Dijkstra's algorithm with a Fibonacci Heap [FT87] and the Bellman-Ford algorithm [Be58].

2.2 Closeness Centrality

Closeness centrality ranks the centralities of graph vertices by their average geodesic distance to all other vertices in a graph [Fr78]. Given two functions $reachable(v)$ that determines the set of vertices reachable from v, and $distance(v, u)$ which determines the geodesic distance from v to u, the normalized closeness centrality of a vertex v is defined as:

$$CC_v = \frac{|reachable(v)|^2}{(|V| - 1) * (\sum_{u \in reachable}(v) : distance(v, u))}$$

2.3 Betweenness Centrality

Betweenness centrality considers a vertex as central when it is on many shortest paths [Fr78]. In an undirected graph, the normalized betweenness centrality of v is defined as:

$$BC_v = \sum_{u,w \in V,\ u \neq v \neq w} : \frac{|\{\mathcal{P} \mid \mathcal{P} \in shortest\_paths(u, w) \wedge v \in \mathcal{P}\}|}{|shortest\_paths(u, w)| * (|reachable(v)|) * (|reachable(v)| - 1)}$$

Here, *shortest_paths*(u, w) denotes the set of all shortest paths $\mathcal{P}$ from u to w. The result is normalized by dividing the share of paths that v is on by the number of vertex pairs in the connected component that do not include v. Note that for undirected graphs the normalization factor has to be adapted to not include duplicate pairs. Naive implementations of betweenness centrality have a runtime in $O(|V|^3)$, even for unweighted graphs. In this paper, we use Brandes's algorithm to compute vertices' betweenness centrality, which reduces the runtime to $O(|V| * |E|)$ for unweighted graphs, and to $O(|V| * |E| + |V|^2 * \log |V|)$ for weighted graphs.

2.4 Multi-Source BFS

In [Th14] the batched multi-source breadth-first search MS-BFS was proposed as an efficient way of solving the geodesic distance problem and to compute vertices' closeness centrality in unweighted graphs. We use MS-BFS as the base of our efficient batched closeness and betweenness centrality algorithms on unweighted graphs, and extend the algorithm's ideas to general batched centrality computation on weighted graphs. In the following, we give an overview of MS-BFS.

MS-BFS runs BFSs from multiple source vertices at the same time and merges traversals that would happen redundantly in independent BFS runs. It is especially beneficial in small-world networks—graphs that have a small diameter and vertex degrees that follow a power-law distribution, e.g., social networks. In small-world networks highly-connected hub vertices are often discovered by multiple concurrent BFSs in the same BFS step, i.e., in the same distance from their respective source. MS-BFS leverages that these BFSs will be very similar for the remainder of their traversals.

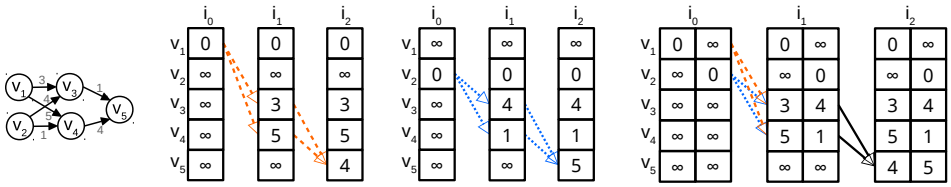
MS-BFS represents the vertices' traversal statuses in multiple concurrent BFSs as bitsets. On these bitsets, the BFS steps are processed concurrently for multiple traversals by means of bit operations, which also implicitly merge traversals. For example, when visiting a vertex v, MS-BFS determines all concurrent traversals that discover its neighbor n in the next BFS step by intersecting the bitset of traversals for which v is visited with the negation of the bitset of traversals that have already visited n. Thus, using SIMD instruction, MS-BFS can concurrently process BFS steps in hundreds of traversals.

3 Batched Algorithm Execution

MS-BFS shares data accesses and computations in multiple BFSs. While this allows for great speedups, MS-BFS is too restrictive. Redundant computations and data accesses do not only occur in multi-source BFSs, but also in many other analytical graph algorithms when they process a graph multiple times. Thus, we propose batched algorithm execution which generalizes the ideas of MS-BFS.

In *batched algorithm execution* an algorithm is redesigned so that it can be efficiently executed concurrently from multiple source vertices. The *concurrent executions* are synchronized to

share common computations and data accesses. Batched algorithms, thus, can amortize the cost of random data accesses over all concurrent executions, leading to greatly reduced overall runtimes. Furthermore, batched algorithm execution facilitates vectorization because the vertices' and edges' associated properties can be processed at the same time for multiple executions.



(a) Graph G (b) Non-batched execution (c) Batched execution

Fig. 1: Data access pattern of the Bellman-Ford algorithm using different executions.

Figure 1 shows how non-batched and batched execution run the Bellman-Ford algorithm. The Bellman-Ford algorithm is executed from the vertices v_1 and v_2 in the small weighted graph G, shown in Figure 1a, to determine their geodesic distances to all vertices. Figure 1b shows how the algorithm is run in traditional non-batched execution by depicting the algorithm's distances array in each iteration. The algorithm is first run from v_1, as shown on the left side: it is initialized in iteration i_0, in i_1 the distances to v_3 and v_4 are discovered, and in iteration i_2 the distance to v_5 is found. Afterward, the algorithm is run again, this time from v_2, as shown on the right side of Figure 1b. The two algorithm executions from v_1 and v_2 process the full graph separately. Consequently, all random accesses, e.g., to determine the current distances of a vertex's neighbors, as well as the resulting memory stalls happen separately for both algorithm executions.

In contrast, a batched variant of the Bellman-Ford algorithm, which we also propose in this paper, can compute the distances from multiple sources at the same time. Figure 1c depicts how our batched algorithm accesses the data when concurrently computing the distances from v_1 and v_2. The first algorithm iteration is still similar to the non-batched case, as no graph and data accesses can be shared among the execution from v_1 and v_2. The batched execution does, however, already show improved data locality. In the second iteration the two algorithm executions share their graph as well as their data accesses. Instead of separately reading each vertex's current distance and comparing it with its neighbor's distances, the two executions from v_1 and v_2 do so at the same time. This allows the batched algorithm to not only amortize its memory access latencies, but also to reduce its memory bandwidth requirements.

In this paper we refer to the *set of concurrent executions* as S, and to the actual concurrent executions as $s_1, \ldots, s_\omega \in S$. We assume that there is a bijection between the concurrent executions and their source vertices; by convention we use the same subscript to associate them. In this section's Bellman-Ford example, there are $\omega = 2$ executions $S = \{s_1, s_2\}$ from v_1 and v_2, respectively.

In batched algorithm execution it is important that accesses from multiple concurrent executions to the same graph property have spatial locality, i.e., data accesses in close temporal succession should operate on memory locations that are close together so that the CPU's caches and prefetchers can be leveraged. To achieve spatial locality we introduce a *batched execution-optimized memory layout* in which all executions' values for variables as well as for vertex and edge properties are co-located, as shown in Figure 1c. We refer to variables with co-located values as *batch variables* and denote a batch variable B of type $\mathcal{T}$ as *BatchVar<*$\mathcal{T}$*> B*. It is stored as a $\mathcal{T}$ array with as many elements as there are concurrent executions of the batched algorithm. Consider that the batched Bellman-Ford algorithm shown in Figure 1c stores the concurrent executions' distances for each vertex in a property *dists* of type *BatchVar<float>*. When the algorithm is executed concurrently from 16 sources, each vertex v's *dists* variable is a 16-element array in which the i^{th} element contains the distance of v in execution i. Assuming that *float*s are 4 byte-wide, v's *dists* field fills a 64-byte cache line—a common size in many modern CPUs. Hence, all concurrent executions' *dists* variable values are in the cache at the same time and no cache space is wasted, e.g., for not required vertices' property values.

4 Distances and Centralities in Unweighted Graphs

In this section we show how batched algorithm execution can be applied to efficiently compute distances and centralities in unweighted graphs. Section 4.1 shortly discusses batched BFS-based distance computation, Section 4.2 describes a batched closeness centrality algorithm with highly efficient constant-time vertex counting, and Section 4.3 presents a novel batched betweenness centrality algorithm.

4.1 Distance

In unweighted graphs the geodesic distance from a source vertex to all other vertices in the same connected component can efficiently be determined using breadth-first search (BFS). In the context of centrality computations, we are interested in computing distances from multiple sources. Thus, we can apply the MS-BFS algorithm and directly write the distance of every discovered vertex into a vertex property that uses our batch-optimized memory layout. Modern CPUs and compute accelerators provide scatter instructions which allow us to write multiple distance assignments in a single instruction. As batched geodesic distance computation in unweighted graphs is already discussed in [Th14] we omit the full algorithm listing.

4.2 Closeness Centrality

A vertex's closeness centrality (CC) is its average geodesic distance to all other vertices. The CC values of a set of vertices S can, thus, be computed by averaging the results from the previous section's batched distance algorithm for every source $s \in S$. There is, however,

no need to actually store the distances of the discovered vertices. Instead, it is sufficient to run a BFS from every source s and directly sum the distances to all discovered vertices per concurrent execution. Listing 1 shows our batched CC algorithm that computes the metric for a set of *vertices* in the unweighted graph G and stores its results in the vertex property *cc*.

```
1  INPUT:  Graph G, Array<Vertex> vertices
2  OUTPUT: VertexProperty<double> cc
3
4  BatchVar<int> iterationVertices=0, totalVertices=0, distanceSums=0
5
6  G.MS-BFS( sources: vertices,
7            onDiscovered: (vertex, discoveredExecutions) => {
8               FOR EACH i in discoveredExecutions:
9                  iterationVertices[i] ++
10           },
11           onIterationEnd: (iteration) => {
12              distanceSums      += iterationVertices * iteration
13              totalVertices     += iterationVertices
14              iterationVertices = 0
15           } )
16
17 FOR i = 1 .. vertices.length:                          ◁ Normalize CC values
18    cc[vertices[i]] = (totalVertices[i] * totalVertices[i])
19                    / (distanceSums[i] * (G.num_vertices-1))
```

List. 1: Batched closeness centrality algorithm for unweighted graphs

The algorithm leverages the inherent property of BFS traversal that all vertices discovered in the same iteration have the same distance to its source. So, rather than summing the actual distances directly, we count the vertices found in each iteration *iterationVertices*, and at the iteration's end add the number of vertices multiplied by the current distance to the actual sum of distances *distanceSums*. We elaborate on counting the number of discovered vertices in the subsequent section. In the algorithms last Lines 17 through 19 we compute the final, normalized CC values, as described in Section 2.2.

4.2.1 Efficient Batch Incrementer

Naively, incrementing *iterationVertices* for every concurrent execution that found a new vertex involves a loop over all executions, as is shown in Lines 8 and 9 of Listing 1. To determine if an execution found a new vertex, a branch whether or not *discoveredExecutions* is *true* for the respective execution is required. This branch is hard to predict and, thus, not efficient on modern CPUs with deep pipelines. In the following, we propose an *efficient batch incrementer* that allows incrementing *iterationVertices* in a branch-free manner, independent of the number of executions in which vertices were actually discovered.

The basic idea of our batch incrementer is to use a single-byte cache to sum up each execution's *true* entries in the *discoveredExecutions* fields over the course of multiple *onDiscovered* calls. Consider that there are 64 concurrent executions. We store all 64 executions' cache values as a contiguous chunk of memory which we can easily increment by *discoveredExecutions* for all iterations at the same time using bit operations. To do so,

we mask out all but each byte's least significant bit in the *discoveredExecutions* bitset and add these eight bytes (the masked bitset) to the first eight cache bytes. Afterward, we shift *discoveredExecutions* left by one bit, repeat the masking and add this masked bitset to the second eight bytes in the cache. This continues until the bitset was shifted seven times, and all cache bytes were touched. At this point, for each i with $discoveredExecutions[i] = true$, the cache byte $\lfloor i/8 \rfloor + (i \mod 8)$ was incremented.

As cache bytes may overflow after 255 increment operations, we flush it to the actual batch variable *discoveredExecutions* after 254 increments.The cache flush increments the batch variable's values by each execution's cached number. After the flush completes, the i^{th} element of *discoveredExecutions* contains the number of times *iterationVertices* was *true* in execution i. All loops in our batch incrementer have a fixed iteration count independent of the actual number of increments; they gain further speedup by unrolling and vectorization. Using our efficient batch incrementer greatly improves the performance of batched closeness centrality computations in unweighted graphs.

4.3 Betweenness Centrality

The betweenness centrality (BC) value of a vertex is a measure for how many shortest paths it is on. As explained in Section 2.3, the state-of-the-art algorithm to compute BC is Brandes's algorithm [Br01]. In the following, we show an efficient batched variant of Brandes's BC algorithm that leverages a novel MS-BFS variant which allows reverse multi-source BFS traversal and can efficiently determine BFS predecessor relationships. In contrast to previous work [Br01, Ma09], it does not need to explicitly store vertex distance and predecessor information, which greatly improves its data locality.

4.3.1 Reverse MS-BFS

The MS-BFS algorithm is designed to discover vertices in increasing distance from the sources. To compute vertices' BC using Brandes's algorithm it is, however, necessary to first process all vertices in ascending distance to calculate their σ values, and then in descending distance order to calculate their δ values. Single-source BFS algorithms do this latter *reverse BFS traversal* by collecting all discovered vertices in a stack, which is traversed once the forward traversal is completed. This approach could be applied to MS-BFS as well by introducing a stack of vertex identifiers and *BatchVar<bool>*s. Instead of building a new datastructure during the MS-BFS traversal, it is more efficient to store and leverage the existing per-iteration bitsets of vertices that must be visited—the iterations' *frontiers*.

Storing all iterations' frontiers for the reverse traversal allows not only to reconstruct the BFS traversal order, but also to efficiently determine BFS predecessor relationships as we show in the next section. For small-world networks—the focus of this paper—the space overhead of storing all frontiers is negligible, as these graphs have a low diameter, which means that only few frontiers must be stored. Furthermore, because there are concurrent

BFS executions, the majority of the stored frontier entries are non-empty and would require a similar amount of memory, or even more, when the less versatile stack would be used.

4.3.2 Implicit Vertex Predecessors

Brandes's BC algorithm uses each vertex's predecessors to accumulate dependencies between vertices. The *predecessors* of a vertex v in a BFS traversal from a source s are all vertices u such that there is an edge (u, v) that is on the shortest path from s to v. Brandes's algorithm and its parallelized variant [Ma09] build an explicit list of predecessors for each vertex. One problem of explicitly stored predecessor lists is that they either require runtime memory allocations or significant over-allocation as their size cannot be determined in advance and is only bounded by the number of edges in the graph. This is especially problematic in the multi-source case, as vertices' predecessors are likely to differ between the concurrent executions. Thus, we propose reconstructing vertices' predecessors from the frontiers of previous MS-BFS iterations.

Lemma. *The bitset of executions in which a vertex p is a predecessor of v can be derived from the frontiers of the current iteration iter and the previous iteration iter*-1 *by means of a bitwise* and *operation:*

$$predecessorIn(p, v) = \begin{cases} frontiers[iter-1][p] \;\&\; frontiers[iter][v], & if\ (p, v) \in E \\ \varnothing, & otherwise \end{cases}$$

Proof. Assume there is an edge between p and v. If p was visited in the BFS iteration directly before v, then p must be in the last iteration's frontier and v in that of the current one, so p is v's predecessor. If p was visited, but v was not visited in the subsequent iteration, the bit operation results in p not being v's predecessor. Similarly, if v is marked in the frontier, but p not in the previous one, p is not v's predecessor. In case there is no edge between p and v, the former cannot be the latter vertex's predecessor, so the set of executions in which p is the predecessor is empty. □

4.3.3 Batched Betweenness Centrality

Building on our proposed reverse MS-BFS and the implicitly-defined vertex predecessors, Listing 2 shows our batched betweenness centrality algorithm for unweighted graphs. It closely follows the structure of Brandes's algorithm, but runs the algorithm from multiple vertices at the same time and batches its execution. Whenever a vertex v is visited in the same distance by multiple concurrent executions, all executions process v at the same time. This is especially beneficial for the complex numeric computations in Lines 17 and 18, as our batch-optimized data layout improves the algorithm's spatial locality and facilitates the use of wide vectorized instructions, allowing multiple executions' computation in the same instruction. Furthermore, batched execution allows to pre-aggregate the *delta* values in

deltaSum before they are added to the global *bc* property; this avoids congestion as multiple parallel threads may access this value.

```
1  INPUT:  Graph G
2  OUTPUT: VertexProperty<double> bc
3
4  VertexProperty<BatchVar<double>> sigma = 0, delta = 0
5  FOR i = 1 .. G.num_vertices:
6      sigma[G.vertices[i]][i] = 1
7
8  G.MS-BFS( sources: G.vertices,
9            onDiscovered: (v, discoveredIn) => {
10               FOR EACH n in G.neighbors(v):
11                   FOR EACH i in (predecessorIn(v,n) & discoveredIn):
12                       sigma[n][i] += sigma[v][i]
13           },
14           onReverse: (v, discoveredIn) => {
15               FOR EACH n in G.neighbors(v):
16                   FOR EACH i in (predecessorIn(v,n) & discoveredIn):
17                       delta[v][i] += (sigma[v][i] / sigma[n][i])
18                                    * (delta[n][i] + 1)
19
20               double deltaSum = 0
21               FOR i = 1 .. G.vertices.num_vertices:
22                   IF v != G.vertices[i]:
23                       deltaSum += delta[v][i]
24               bc[v] += deltaSum
25           } )
26
27 FOR EACH v in G.vertices:                          ◁ Normalize BC values
28     bc[v] /= (G.num_vertices - 1) * (G.num_vertices - 2)
```

List. 2: Batched betweenness centrality algorithm for unweighted graphs

Note that in contrast to the previously shown closeness centrality algorithm that can selectively be run for a subset of the vertices to determine their centrality, BC must always process all vertices in the graph—or more specifically, in a given connected component—to compute the metric. All presented techniques can also be applied to approximate BC algorithms [Ba07].

For a lack of space we do not give details about the parallelization of our algorithm. Our approach is, however, similar to the parallelization presented in [Ma09].

5 Distances and Centralities in Weighted Graphs

In this section we focus on batched distance and centrality computation on weighted graphs. We first discuss batched weighted multi-source geodesic distance computation in Section 5.1, as it is an important building block for centrality computation. Afterward, in Sections 5.2 and 5.3, we show how the calculated distances can be used to efficiently derive the closeness and betweenness centrality metrics, respectively.

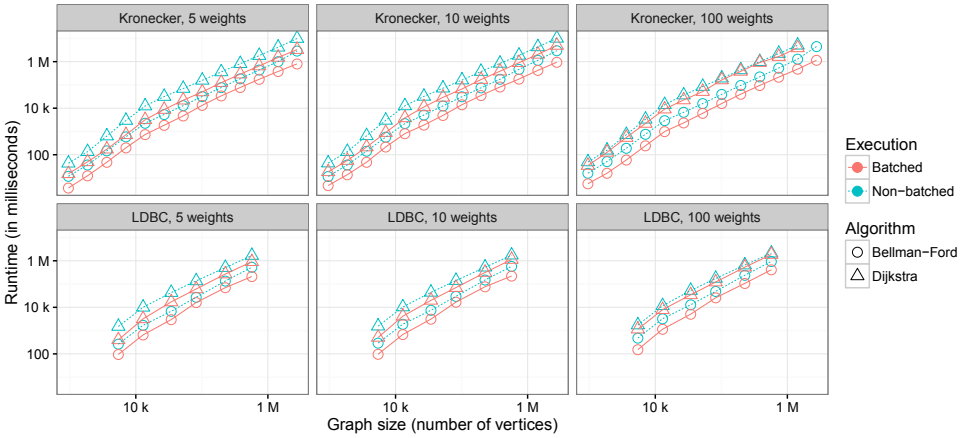


Fig. 2: Comparison of multi-source shortest geodesic distance runtimes for non-batched and batched variants of Dijkstra's algorithm and the Bellman-Ford algorithm, applied to Kronecker and LDBC graphs of various sizes and with different weight counts.

5.1 Distance

In practice, two algorithms are commonly used to compute the distances from one or more vertices to all other vertices in a weighted graph: Dijkstra's algorithm with a Fibonacci heap [FT87], and the Bellman-Ford algorithm [Be58]. While the worst-case runtime of the Bellman-Ford algorithm is $O(|V| * |E|)$, previous work [Ye70] shows that it has an expected runtime of $O(E)$ in large dense graphs with low diameter. We, thus, analyzed the runtimes of the algorithms using LDBC and Kronecker small-world networks, which we also used in our evaluation. We generated edge weights in the range $[1, \mathit{weightcount}]$ for three weight counts: 5, 10 and 100; the former two are commonly used in datasets with user-generated ratings, and the latter indicates the algorithms' scalability with higher weight counts. Figure 2 shows our results for normal and batched variants (solid and dashed lines, respectively) of both Dijkstra's algorithm and the Bellman-Ford algorithm (triangles and circles, respectively). While both algorithms show similar scaling behavior with increasing graph size, the Bellman-Ford algorithm's absolute performance is 3-10× higher. In all measurements, the batched algorithm variants show better runtimes than the respective non-batched algorithm. Virtually unaffected by the used weight count, the Bellman-Ford algorithm's batched variant is 3-4× faster than the non-batched algorithm. In contrast, the benefit of our batched Dijkstra's algorithm becomes smaller for higher weight counts.

We only discuss the details of our batched multi-source Bellman-Ford algorithm as it—and even its original single-source variant—clearly outperforms Dijkstra's algorithm for the evaluated small-world networks with even hundreds of different weights. Listing 3 depicts our proposed algorithm. It expects a weighted graph G, and a set of *sources* from which the distances to all other vertices should be calculated. The results are stored in a batch vertex property *dists*, that contains the vertices' distance from each of the sources.

Listing 3 has the same structure as the original Bellman-Ford algorithm. It initializes the sources' distances with 0 and all other vertices' distances with infinity, and iteratively checks whether new shorter distances can be found based on the already known distances. Unlike the original algorithm, our batched multi-source Bellman-Ford algorithm discovers new shortest distances for *all* sources' executions at the same time. As a result, the executions can share the random data accesses to the neighbors' known distances, amortizing the memory access costs over all concurrent executions. Furthermore, our batched algorithm can leverage the wide vector instructions of modern compute accelerators like the Intel Xeon Phi, e.g., for the distance computation in Line 20.

We use an optimization proposed by Yen [Ye70] and only check vertices' neighbors if their distance was modified. For this we introduce a batch vertex property *modified* that keeps track of modified vertices and the executions in which their distance was updated. For simplicity, the depicted algorithm uses one *modified* entry per concurrent execution. Our actual implementation uses one entry per i executions, where i is the number of distance values that can be stored and processed simultaneously in the target CPU's largest SIMD register. Using one *modified* entry for multiple executions avoids branches in the algorithm's hot loop and has only negligible overhead for distance computations that are done unnecessarily as all operations are vectorized. When no distance was *changed* in an iteration of the algorithm, no shorter geodesic distances may be found and the algorithm finishes.

```
1  INPUT:  WeightedGraph G, Array<Vertex> sources
2  OUTPUT: VertexProperty<BatchVar<double>> dists
3
4  VertexProperty<BatchVar<bool>> modified = false
5  dists = Infinite
6
7  FOR i = 1 .. sources.length:
8      Node v = sources[i]
9      dists[v][i]    = 0
10     modified[v][i] = true
11
12 bool changed = true
13 WHILE changed:
14     changed = false
15     FOR EACH v in G.vertices:
16         IF not modified[v].empty():
17             FOR EACH n in G.neighbors(v):
18                 double weight = edgeWeight(v,n)
19                 FOR EACH i in modified[v]:
20                     double newDist = min(dists[n][i], dists[v][i] + weight)
21                     IF newDist != dists[n][i]:
22                         dists[n][i]    = newDist
23                         modified[n][i] = true
24                         changed        = true
```

List. 3: Batched Bellman-Ford-based geodesic distance algorithm for weighted graphs

5.2 Closeness Centrality

Based on the batched multi-source geodesic distance computation described in the previous section, we propose the efficient batched closeness centrality (CC) algorithm shown in Listing 4. For a given weighted graph G, it computes the exact CC values of a set of *vertices* and stores them as the vertex property *cc*.

Our batched CC algorithm first computes the distances from the *vertices* of interest to all other vertices in their respective connected components using the batched distance computation presented in the previous section. It then counts the reachable vertices *totalVertices* and sums their distances *distanceSums* concurrently for all executions. Because of the data layout of the batched variables *totalVertices* and *distanceSums*, this computation can be automatically vectorized by modern compilers. At the end of the algorithm, the final CC values are normalized, as described in Section 4.2.

```
1  Input:  WeightedGraph G, Array<Vertex> vertices
2  Output: VertexProperty<double> cc
3
4  VertexProperty<BatchVar<double>> dists
5  dists = ms_geodesic_distances(G, vertices)        ◁ Listing 3
6  BatchVar<double> distanceSums = 0
7  BatchVar<int> totalVertices   = 0
8
9  for each v in G.vertices:
10   for i = 1 .. vertices.length:
11     if dists[v][i] != Infinite:                   ◁ Vertex is reachable
12       distanceSums[i]  += dists[v][i]
13       totalVertices[i] ++
14
15 normalize(G, vertices, cc)                        ◁ Normalize CC, see Listing 1
```

List. 4: Batched closeness centrality algorithm for weighted graphs

5.3 Betweenness Centrality

In the following, we present a novel batched betweenness centrality (BC) algorithm for weighted graphs that is again based on Brandes's algorithm.

5.3.1 Batched Distance Ordering

A batched variant of Brandes's BC algorithm must find a global order of traversing a weighted graph such each execution traverses the graph with ascending distance from its respective source. An efficient global order further ensures that the executions share as many computations and data accesses as possible. Our batched BC algorithm builds such a global order using a hash-based approach that determines for each vertex-distance pair (v, d) the set of executions $T \subseteq S$ which discover v in distance d. It then sorts the resulting

(d, v, T)-tuples by their distance d and merges tuples when this is possible without violating the ordering.

Our approach does not yet consider possible tuple reorderings that do violate the strict distance order without impacting the algorithm's result. Assume there are three tuples o_1, o_2, o_3 with $o_i = (d_i, v_i, T_i)$ and $d_1 < d_2 < d_3$. When $v_1 = v_3$ and $T_1 \cap T_2 \cap T_3 = \emptyset$, o_1 and o_3 can safely be merged to allow improved execution sharing. There is, however, a tradeoff between preprocessing time to determine an optimal order and the actual algorithm execution time. Exploring this tradeoff and finding heuristics for efficient reordering are interesting directions for future work.

5.3.2 Batched Betweenness Centrality

We propose the batched algorithm shown in Listing 5 to efficiently compute the BC values for all vertices in a weighted graph G and store them in a vertex property bc. It computes the geodesic distances for all vertices using the batched Bellman-Ford-based algorithm proposed in Section 5.1 and builds a global traversal order in which Brandes's algorithm is executed for multiple executions concurrently, sharing common data accesses and computations.

```
1  Input:  WeightedGraph G
2  Output: VertexProperty<double> bc
3
4  VertexProperty<BatchVar<double>> dists
5  dists = ms_geodesic_distances(G, G.vertices)        ◁ Listing 3
6  List<Tuple<double, Node, ExecutionSet>> traversalOrder
7  traversalOrder = findOrdering(G, dists)             ◁ Section 5.3.1
8  initialize(sigma, delta)                            ◁ See Lines 4-6 in Listing 2
9
10 for each (d,v,T) in traversalOrder:
11     for each n in G.neighbors(v):
12         for each s in T:                            ◁ Executions with v in distance d
13             if dists[n][s] == d + edgeWeight(v,n):
14                 sigma[v][s] += sigma[n][s]          ◁ Vertex is a predecessor
15
16 for each (d,v,T) in traversalOrder.reverse():
17     for each n in G.reverseNeighbors(v):
18         for each s in T:
19             if dists[n][s] + edgeWeight(n,v) == d:
20                 delta[n][s] += sigma[n][s]*(1+delta[v][s])/sigma[v][s]
21         addBC(bc[v], delta[v], s)                   ◁ See Lines 20-24 in Listing 2
22
23 normalize(G, bc)                                   ◁ See Lines 27-28 in Listing 2
```

List. 5: Batched betweenness centrality algorithm for weighted graphs

While the structure of the algorithm is then similar to our batched unweighted BC algorithm, it differs in two important ways: One, instead of using the BFS-defined traversal order, Listing 5 uses the optimized batch distance ordering *traversalOrder* for both the forward and reverse traversal. Two, as explained for the batched unweighted BC, it is unfeasible to explicitly store the predecessors of all vertices in all concurrent executions. Our weighted

BC algorithm does, however, not reconstruct each vertex v's predecessors from the traversal history like the former algorithm does, but determines them directly by checking the neighbors' distances to v: when the neighbor n's distance plus the incident edge's weight equals v's distance, then n must be on the shortest path to v, and, hence, its predecessor.

Note that for simplicity, the depicted algorithm computes the *sigma* and *delta* values for all vertices in the graph at the same time. In an implementation that is suited to scale to large real-world graphs, the algorithm's executions for Lines 4 through 21 must be run using smaller batches of executions, such that only a subset of the source vertices is considered at a time [Th14].

6 Evaluation

In this section we evaluate the runtime and scalability of the algorithms we propose in this paper. We first give a short description of our experiment setup. Next, we evaluate how the batch size—the number of concurrently processed executions—influences the efficiency of batched algorithm execution. Afterward, we discuss how our centrality algorithms scale with increasing graph size; we omit the distance algorithms' scalability, as it was already discussed in Section 5.1.

6.1 Experiment Setup

To evaluate the efficiency of the batched algorithms proposed in this paper, we implemented them as standalone C++14 programs and compiled them using GCC 5.2.1. We derived our MS-BFS implementation from the original authors' provided sources [Th14]. As competitors we used non-batched variants of the algorithms and optimized them according to the state-of-the-art. For unweighted and weighted betweenness centrality we additionally ported Brandes's implementation, kindly provided by the author [Br01], to the graph structures used in all other implementations.

We evaluated all algorithms using synthetic and real-world datasets of various sizes. We obtained the Citeseer (384k vertices), DBLP (1.3M vertices), Hudong (3M vertices) and Wikipedia (1.9M vertices) real-world graphs datasets from the KONECT repository [Ku13]. The synthetic LDBC and Kronecker graphs we used comprised up to 4.2M vertices and 300M edges. Linked Database Counsil (LDBC) graphs are designed to resemble real-world social networks [Io16]. We generated them using the LDBC SNB generator version 0.2.6[5]. Kronecker graphs are used in the common graph benchmark Graph500 and were built using the benchmark's data generator[6] with edge factor set to 32.

All experiments were executed on a dual-socket system equipped with Intel Xeon E5-2660 v2 CPUs (20 logical threads at 2.2GHz) and 256GB of main memory. The used operating system was Ubuntu Linux 15.10 with kernel 4.2. To reduce the experiments' runtimes, we ran all algorithms for 15,360 deterministically random selected vertices in the graphs.

[5] https://github.com/ldbc/ldbc_snb_datagen
[6] https://github.com/graph500/graph500

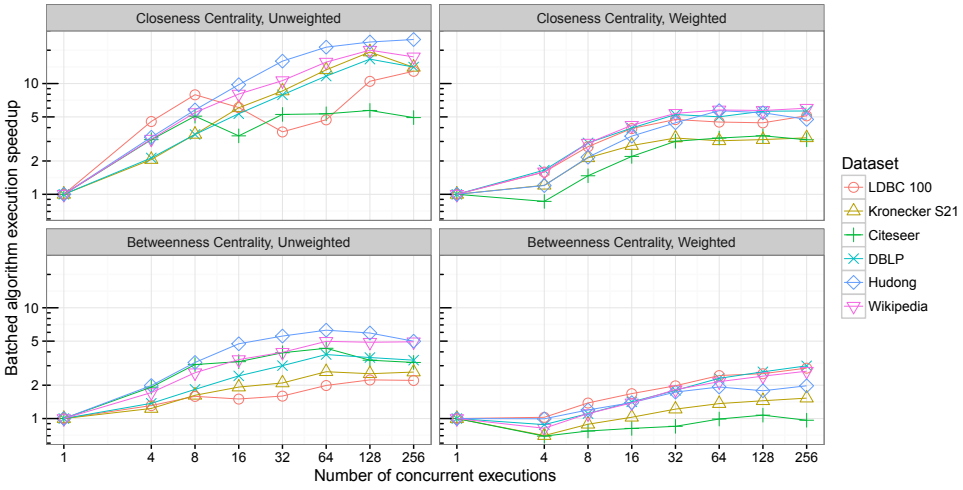


Fig. 3: Speedup through batched algorithm execution with increasing number of concurrent executions.

6.2 Scalability with Number of Concurrent Executions

This paper proposes batched algorithm execution for geodesic distance and centrality computation. In the following we show how batching influences the runtime of the presented algorithms. For each algorithm we measured the non-batched runtime as well as the absolute runtimes for increasing numbers of concurrent executions. For weighted graphs we assume a weight count of five. Figure 3 shows the speedups gained through batched algorithm execution over non-batched execution.

It can be seen that batching leads to significant speedups, even for few concurrent executions. The more concurrent executions are used, the higher the speedup, because more executions can share graph and data accesses. For unweighted graphs, batched closeness centrality shows 5-11× speedup over non-batched execution, depending on the analyzed graph. The significantly more complex batched betweenness centrality algorithm exhibits 2-7× speedup on unweighted graphs.

Both algorithms' speedups are a result of the amortized memory access costs achieved by batched execution. For weighted graphs our batched weighted closeness centrality algorithm shows between 3 and 6× better performance than the respective state-of-the-art non-batched algorithm. Batched betweenness centrality on weighted graphs shows up to 3× speedup. Our measurements show, however, that batching can also have a negative impact on execution performance for low numbers of concurrent executions. This is the case when the additional computation added by batched processing cannot be amortized by the savings from its improved memory access pattern. When enough concurrent executions exist, the speedups of our weighted algorithms are again caused by the amortized memory access costs, but also by the increased numeric throughput achieved by using SIMD instructions during the geodesic distance computation.

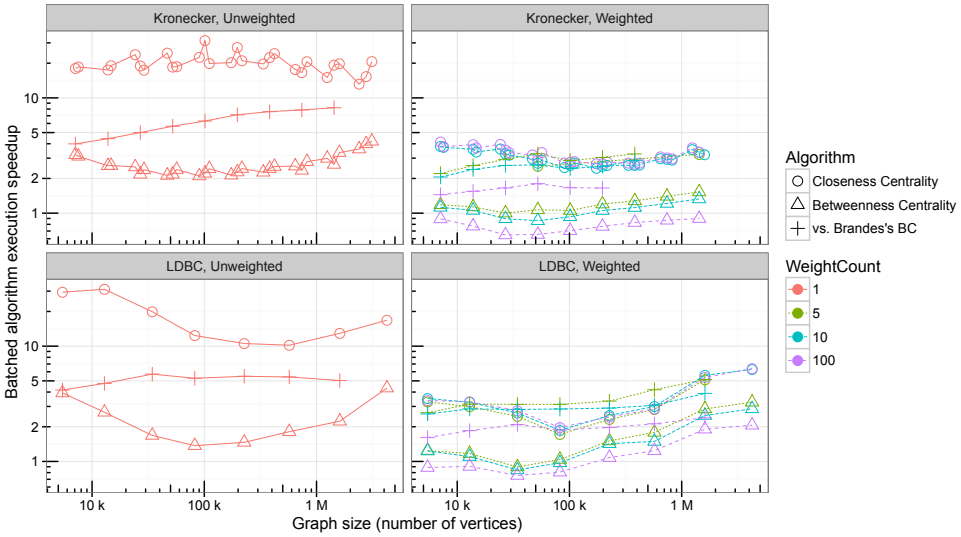


Fig. 4: Speedup of batched algorithms for Kronecker and LDBC graphs of various sizes.

6.3 Graph Size Scalability

Batched algorithm execution is designed to amortize the cost of random data accesses. While random accesses do not significantly impact small graphs with less than a million edges—as such graphs easily fit into modern CPU's caches—for large real-world graphs they are likely to cause cache misses, which lead to high-latency main memory accesses and CPU stalls. We evaluate the speedup of our proposed batched algorithms over non-batched implementations for Kronecker and LDBC graphs of various sizes in Figure 4. The figure shows the speedups of our batched closeness centrality and betweenness centrality algorithms over non-batched execution as circles and triangles, respectively, and depicts our speedup compared to Brandes's implementation using crosses. All measurements were done for graphs with various counts of edge weights, which we represent as different colors and line types.

Our measurements exhibit the expected effects: When small graphs are analyzed, both the graph and the algorithm's working set fit into the CPU cache. In this situation, batched execution benefits mostly from vectorization and avoided redundant computation in the concurrent executions. For medium-size graphs, the speedup of our batched algorithms reduces slightly. The reason for this is that because batched algorithms run multiple executions concurrently, they have a larger working set than non-batched algorithms and outgrow the CPU cache faster. For large graphs, the speedup of batched execution increases again. At this point, both the batched and non-batched algorithms' graph data and working sets do not fit into the CPU cache anymore, but only batched algorithms efficiently amortize the cost of main memory accesses. While both the Kronecker and the LDBC graphs exhibit the expected behavior, Figure 4 shows that it is more pronounced for the LDBC graph.

The achieved speedups are similar for Kronecker and LDBC graphs of similar sizes as we already discussed in the previous section. Compared to Brandes's implementation, our batched betweenness centrality algorithms show significant speedups of around 5× for both datasets in the unweighted case or with few different weights.

6.4 Edge Weight Count Scalability

We further evaluated how the number of different edge weights influences the batched algorithms' speedup. Figure 4 shows that our weighted closeness centrality algorithm is nearly independent of the number of different weights. Its runtime is dominated by the batched Bellman-Ford shortest distance computations, which in turn is mostly influenced by the graph's diameter, as discussed in Section 5.1.

In contrast, our weighted betweenness centrality algorithm is noticeably influenced by the weight count. While it also uses the batched Bellman-Ford distance algorithm, its runtime is dominated by the forward and reverse traversal in the graph, which allows less sharing when more different weights are used. A consequence of this reduced sharing is that for very high weight counts our speedup over Brandes's algorithm become less pronounced.

7 Related Work

For decades there has been research on the geodesic distance problem, which is the basis for closeness and betweenness centrality calculations. We focus on distances in small-world networks—low-diameter graphs with a degree distribution that follows the power law. To calculate geodesic distances in unweighted and non-indexed small-world networks we build on the MS-BFS algorithm [Th14]. Kaufmann et al. [Ka17] recently proposed the highly-optimized parallelized MS-PBFS which is orthogonal to our algorithms and can be used to improve the performance of our MS-BFS-based unweighted centrality algorithms.

For weighted non-indexed small-world networks we confirmed [Ye70] that the Bellman-Ford algorithm is very efficient and furthered this work with our batched Bellman-Ford algorithm. This is the first work that proposes batched execution for geodesic distance calculation in dense weighted graphs. In contrast to techniques that are designed for general graphs like Thorup et al. [Th04], our algorithms are optimized for the specifics of dense graphs and the characteristics of modern systems with long memory access latencies. However, while our algorithms optimize for the properties of dense small-world networks, they work on general graphs and do not require special properties like planarity [Kl05]. Furthermore, our work does not use prior graph indexing, as is done in [AIY13], but may be suited to speed up existing indexing techniques.

Once all geodesic distances from a vertex are known, this vertex's closeness centrality value can be computed. Because exact closeness calculation is very expensive in large real-world graphs, approximate algorithms [EW01] and heuristics to find the top-k vertices with the highest closeness centrality values [OLH14] were proposed. Our batched unweighted

closeness centrality algorithm is orthogonal to these approximations and heuristics, and can be used to improve their performance. Building on Brandes's algorithm [Br01], parallelized exact betweenness centrality algorithms [Ma09] as well as approximations [Ba07] have been proposed. The concepts of our batched betweenness centrality algorithm can be applied to further improve their performance. We found existing work on centralities to only explicitly cover the case of unweighted graphs. Weighted graphs are seen as a trivial extension and only briefly mentioned. This is the first work to explicitly evaluate the tradeoffs and optimizations of weighted betweenness centrality and how batching can be applied to it.

8 Conclusion

Batched algorithm execution significantly improves the runtime of distance, closeness centrality, and betweenness centrality computation on unweighted and weighted graphs. In future work we want to find further algorithms that are suited for batched execution.

9 Acknowledgments

This research was supported by the German Research Foundation (DFG), Emmy Noether grant GU 1409/2-1, and by the Technical University of Munich - Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement no 291763, co-funded by the European Union. Manuel Then is a recipient of the Oracle External Research Fellowship. Part of this work was conducted during an internship at Oracle Labs.

References

[AIY13] Akiba, Takuya; Iwata, Yoichi; Yoshida, Yuichi: Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, pp. 349–360, 2013.

[Ba07] Bader, David A; Kintali, Shiva; Madduri, Kamesh; Mihail, Milena: Approximating Betweenness Centrality. In: International Workshop on Algorithms and Models for the Web-Graph. Springer, pp. 124–137, 2007.

[Be58] Bellman, Richard: On a Routing Problem. Quarterly of applied mathematics, pp. 87–90, 1958.

[BP98] Brin, Sergey; Page, Larry: The Anatomy of a Large-Scale Hypertextual Web Search Engine. In: WWW. pp. 3825–3833, 1998.

[Br01] Brandes, Ulrik: A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology, 25:163–177, 2001.

[EW01] Eppstein, David; Wang, Joseph: Fast Approximation of Centrality. In: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, pp. 228–229, 2001.

[Fr78] Freeman, Linton C: Centrality in Social Networks Conceptual Clarification. Social networks, 1(3):215–239, 1978.

[FT87] Fredman, Michael L.; Tarjan, Robert Endre: Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. J. ACM, 34(3):596–615, July 1987.

[Ho15] Hong, Sungpack; Depner, Siegfried; Manhardt, Thomas; Van Der Lugt, Jan; Verstraaten, Merijn; Chafi, Hassan: PGX. D: a fast distributed graph processing engine. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, p. 58, 2015.

[Io16] Iosup, Alexandru; Hegeman, Tim; Ngai, Wing Lung; Heldens, Stijn; Prat, Arnau; Manhardt, Thomas; Chafi, Hassan; Capota, Mihai; Sundaram, Narayanan; Anderson, Michael et al.: LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. Proceedings of the VLDB Endowment, 9(12), 2016.

[Ka17] Kaufmann, Moritz; Then, Manuel; Kemper, Alfons; Neumann, Thomas: Parallel Array-Based Single- and Multi-Source Breadth First Searches on Large Dense Graphs. In: EDBT. 2017.

[Kl05] Klein, Philip N: Multiple-Source Shortest Paths in Planar Graphs. In: SODA. volume 5, pp. 146–155, 2005.

[Ku13] Kunegis, Jérôme: Konect: The Koblenz Network Collection. In: Proceedings of the 22nd international conference on World Wide Web companion. International World Wide Web Conferences Steering Committee, pp. 1343–1350, 2013.

[Ma09] Madduri, Kamesh; Ediger, David; Jiang, Karl; Bader, David A; Chavarria-Miranda, Daniel: A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets. In: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, pp. 1–8, 2009.

[Ma10] Malewicz, Grzegorz; Austern, Matthew H; Bik, Aart JC; Dehnert, James C; Horn, Ilan; Leiser, Naty; Czajkowski, Grzegorz: Pregel: A System for Large-Scale Graph Processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, pp. 135–146, 2010.

[NSJ11] Ni, Chaoqun; Sugimoto, Cassidy; Jiang, Jiepu: DegreE, Closeness, and Betweenness: Application of Group Centrality Measurements to Explore Macro-Disciplinary Evolution Diachronically. In: Proceedings of ISSI. pp. 1–13, 2011.

[OLH14] Olsen, Paul W.; Labouseur, Alan G.; Hwang, Jeong-Hyon: Efficient Top-k Closeness Centrality Search. In: 2014 IEEE 30th International Conference on Data Engineering. pp. 196–207, March 2014.

[Th04] Thorup, Mikkel: Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem. J. Comput. Syst. Sci., 69(3):330–353, November 2004.

[Th14] Then, Manuel; Kaufmann, Moritz; Chirigati, Fernando; Hoang-Vu, Tuan-Anh; Pham, Kien; Kemper, Alfons; Neumann, Thomas; Vo, Huy T.: The More the Merrier: Efficient Multi-source Graph Traversal. Proceedings of the VLDB Endowment, 8(4):449–460, December 2014.

[Ye70] Yen, Jin Y: An Algorithm for Finding Shortest Routes from All Source Nodes to a Given Destination in General Networks. Quarterly of Applied Mathematics, pp. 526–530, 1970.

Streaming and Dataflows

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 269

Gilbert: Declarative Sparse Linear Algebra on Massively Parallel Dataflow Systems

Till Rohrmann[1] Sebastian Schelter[2] Tilmann Rabl[3] Volker Markl[4]

Abstract: In recent years, the generated and collected data is increasing at an almost exponential rate. At the same time, the data's value has been identified in terms of insights that can be provided. However, retrieving the value requires powerful analysis tools, since valuable insights are buried deep in large amounts of noise. Unfortunately, analytic capacities did not scale well with the growing data. Many existing tools run only on a single computer and are limited in terms of data size by its memory. A very promising solution to deal with large-scale data is scaling systems and exploiting parallelism.

In this paper, we propose Gilbert, a distributed sparse linear algebra system, to decrease the imminent lack of analytic capacities. Gilbert offers a MATLAB®-like programming language for linear algebra programs, which are automatically executed in parallel. Transparent parallelization is achieved by compiling the linear algebra operations first into an intermediate representation. This language-independent form enables high-level algebraic optimizations. Different optimization strategies are evaluated and the best one is chosen by a cost-based optimizer. The optimized result is then transformed into a suitable format for parallel execution. Gilbert generates execution plans for Apache Spark® and Apache Flink®, two massively parallel dataflow systems. Distributed matrices are represented by square blocks to guarantee a well-balanced trade-off between data parallelism and data granularity.

An exhaustive evaluation indicates that Gilbert is able to process varying amounts of data exceeding the memory of a single computer on clusters of different sizes. Two well known machine learning (ML) algorithms, namely PageRank and Gaussian non-negative matrix factorization (GNMF), are implemented with Gilbert. The performance of these algorithms is compared to optimized implementations based on Spark and Flink. Even though Gilbert is not as fast as the optimized algorithms, it simplifies the development process significantly due to its high-level programming abstraction.

Keywords: Dataflow Optimization, Linear Algebra, Distributed Dataflow Systems

1 Introduction

Key component of modern big data solutions are sophisticated analysis algorithms. Unfortunately, the search for useful patterns and peculiarities in large data sets resembles the search for a needle in a haystack. This challenge requires tools that are able to analyze vast amounts of data quickly. Many of these tools are based on statistics to extract interesting features.

It is beneficial to apply these statistical means to the complete data set instead of smaller chunks. Even if the chunks cover the complete data set, important correlations between

[1] Apache Software Foundation, trohrmann@apache.org
[2] Technische Universität Berlin, sebastian.schelter@tu-berlin.de
[3] Technische Universität Berlin / DFKI, rabl@tu-berlin.de
[4] Technische Universität Berlin / DFKI, volker.markl@tu-berlin.de

data points might get lost if chunks are treated individually. Moreover, the statistical tools improve their descriptive and predictive results by getting fed more data. The more data is available, the more likely it is that noise cancels out and that significant patterns manifest. However, these benefits come at the price of an increased computing time, which requires fast computer systems to make computations feasible.

Analysis tools traditionally do not scale well for vast data sets. Because of the exponential data growth, analytic capacities have to improve at a similar speed. This problem can be tackled by vertical and horizontal scaling of computers. To scale vertically means that we add more resources to a single computer. For example, the main memory size or the CPU frequency of a computer can be increased to improve computational power. In contrast to that, horizontal scaling refers to adding more computer nodes to a system. By having more than one node, computational work can be split up and distributed across multiple computers.

The emerging field of multi-core and distributed systems poses new challenges for programmers. Since they have to be able to reason about interwoven parallel control flows, parallel program development is highly cumbersome and error-prone. Therefore, new programming models are conceived, a development that relieves the programmer from tedious low-level tasks related to parallelization such as load-balancing, scheduling, and fault tolerance. With these new models, programmers can concentrate on the actual algorithm and use case rather than reasoning about distributed systems. These are the reasons why Google's MapReduce [DG08] framework and its open source re-implementation Hadoop [08] became so popular among scientists as well as engineers.

MapReduce and other frameworks, however, force users to express programs in a certain way which is often not natural or intuitive to users from different domains. Especially, in the field of data analytics and machine learning programs are usually expressed in a mathematical form. Therefore, systems such as MATLAB [Ma84] and R [93] are widely used and recognized for their fast prototyping capabilities and their extensive mathematical libraries. However, these linear algebra systems lack proper support for automatic parallelization on large clusters and are thus restricting the user to a single workstation. Therefore, the amount of processable data is limited to the size of a single machine's memory, which constitutes a serious drawback for real-world applications.

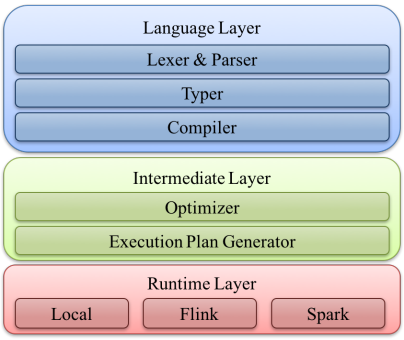


Fig. 1: Gilbert's system architecture consisting of the language, intermediate and runtime layer.

As a solution we propose Gilbert, a distributed sparse linear algebra environment whose name is a homage to William Gilbert Strang. Gilbert provides a MATLAB-like language for distributed sparse linear algebra operations. It has a layered architecture, which is shown in Figure 1. The first layer is the language layer. This layer parses Gilbert code and compiles it into an intermediate representation (IR). The second layer is the intermediate layer, which is given the IR. The intermediate format is the ideal representation to apply language independent high-level transformations. The Gilbert optimizer applies several algebraic optimizations prior to the generation of the execution plan. The execution plan generator translates the optimized IR into an execution engine specific plan. This execution plan is then executed on the respective back end.

Our main contributions are:

- Gilbert allows to write MATLAB-like code for sparse linear algebra and execute it on massively parallel dataflow systems.
- Its expressiveness allows to quickly develop scalable algorithms to analyze web-scale data.
- We introduce a novel fixed-point operator which replaces loops by recursion and can be efficiently parallelized.
- Furthermore, we demonstrate that dataflow optimizers can be used to automatically select a good matrix multiplication strategy.

The rest of this paper is structured as follows. In Sect. 2, Gilbert's language and its features are described. Sect. 3 presents how linear algebra operations are mapped to a dataflow system. Gilbert's performance and scalability is evaluated in Sect. 4. Related work is covered in Sect. 5 before the work is concluded in Sect. 6.

2 Gilbert Features

Gilbert's language has been strongly inspired by MATLAB, which should ease its adoption. The language is a fully functional subset of MATLAB and its grammar is specified in [Ro14]. The elementary data type is the matrix. Gilbert supports arbitrary 2-dimensional matrices whose elements can be *double* or *boolean*. Vectors are not represented by a special type but instead are treated as a single column/row matrix. Additionally, scalar *double*, *boolean*, and *string* values are supported. Gilbert also implements cell array types. A cell array is defined using curly braces and commas to separate individual values. The cells can be accessed by an index appended in curly braces to the cell array variable. List. 1 shows how to define and access them.

Gilbert supports the basic linear algebra operations defined on matrices and scalars. They include among others +, -, /, and *, whereas * denotes the matrix-matrix multiplication and all other operations are interpreted cell-wise. The cell-wise multiplication is indicated by a point preceding the operator. Gilbert also supports comparisons operators such as >,

```
c = {true, 2*2, 'cell', 'array'};
b = c{1} & false; % = false
d = c{2} ^ 2; % = 16
s = {c{4}, c{3}}; % = {'array', 'cell'}
```

Listing 1: Cell array usage in Gilbert. Definition of a 4 element cell array, which is accessed subsequently.

>=, ==, and ~=. Besides the basic arithmetic operations, the user can also define named functions and anonymous functions. The syntax of anonymous functions adheres to the MATLAB syntax: `@(x,y) x*x + y*y`.

An important aspect of MATLAB are loops. MATLAB permits the user to express `for` and `while` loops, which can have side effects. Parallelizing iterations with side effects is difficult because the referenced external state has to be maintained. This circumstance makes preprocessing and execution unnecessarily complex. Gilbert offers a fixpoint operator `fixpoint`, which iteratively applies a given update function f on the previous result of f, starting with an initial value x at iteration 0.

$$n^{th} \text{ iteration} \equiv \underbrace{f(f(\ldots(f(x))\ldots))}_{n \text{ times}}$$

In order to terminate the fixpoint operation, the operator provides two mechanisms. The user has to specify a maximum number `m` of iterations. Additionally, the user can provide a convergence function `c` to the fixpoint operator. The convergence function is called with the previous and current fixpoint value and returns a boolean value. Thus, the fixpoint operator terminates either if convergence was detected or if the maximum number of iterations is exceeded.

$$fixpoint : \underbrace{T}_{\texttt{x}} \times \left(\underbrace{T \to T}_{\texttt{f}} \right) \times \underbrace{\mathbb{N}}_{\texttt{m}} \times \left(\underbrace{T \times T \to \mathbb{B}}_{\texttt{c}} \right) \to T \tag{1}$$

with T being a generic type variable.

In fact, the fixpoint operator replaces iterations by recursions with a pure update function f. At this point Gilbert breaks with existing MATLAB code. However, all MATLAB loops can be expressed via the fixpoint operator by passing the loop's closure to the update function, see List. 2.

```
A = 0;
for i = 1:10
  A = A + i;
end
```

(a) For loop

```
f = @(x) ...
  {x{1} + x{2}, x{2} + 1};
r = fixpoint({0,1}, f, 10);
A = r{1};
```

(b) Fixpoint

Listing 2: Transformation from MATLAB for loop (a) to Gilbert fixpoint (b) formulation. Essentially, all iteration data is combined and passed as a cell array value to the update function.

2.1 Gilbert Typing System

MATLAB belongs to the class of dynamically typed languages. However, the parallel data processing systems used to run Gilbert programs need to know the occurring data types. Therefore, the MATLAB language has to be enriched with type information. We infer this type information using the Hindley-Milner (HM) type system [Hi69; Mi78] and a slightly derived form of algorithm W [DM82] for type inference. In case that the type inference algorithm cannot properly infer the types, there has to be a way to resolve this problem. Similar to [Fu09], we allow to add type information via special comments.

Function Overloading. MATLAB's basic operators, such as +, -, / and *, for example, are overloaded. They can be applied to matrices, scalars as well as mixture of both data types. That makes it very convenient to express mathematical problems, but from a programmer's point of view it causes some hardships. Originally, HM cannot deal with overloaded functions, because it assumes that each function has a unique type. In order to extend HM's capabilities, we allowed each function symbol to have a list of signatures. In the case of +, the list of signatures would consist of

$$\begin{aligned} matrix[double] \times matrix[double] &\rightarrow matrix[double] \\ matrix[double] \times double &\rightarrow matrix[double] \\ double \times matrix[double] &\rightarrow matrix[double] \\ double \times double &\rightarrow double \end{aligned}$$

In order to solve the typing problem, the inference algorithm has to resolve this ambiguity. Having complete knowledge of the argument types is enough to select the appropriate signature. Sometimes even partial knowledge is sufficient.

Matrix Size Inference. Matrices constitute the elementary data type in our linear algebra environment. Besides its element type, a matrix is also defined by its size. In the context of program execution, knowledge about matrix sizes can help to optimize the evaluation. For instance, consider a threefold matrix multiplication $A \times B \times C$. The multiplication can be evaluated in two different ways: $(A \times B) \times C$ and $A \times (B \times C)$. For certain matrix sizes one way might be infeasible whereas the other way can be calculated efficiently due to the matrix size of the intermediate result $(A \times B)$ or $(B \times C)$.

By knowing the matrix sizes, Gilbert can choose the most efficient strategy to calculate the requested result. Another advantage is that we can decide whether to carry out the computation in-core or in parallel depending on the matrix sizes. Sometimes the benefit of parallel execution is smaller than the initial communication overhead and thus it would be wiser to execute the calculation locally. Furthermore, it can be helpful for data partitioning on a large cluster and to decide on a blocking strategy with respect to the algebraic operations. Therefore, we extended the HM type inference to also infer matrix sizes where possible.

Gilbert's matrix type is defined as

$$MatrixType(\underbrace{\tau}_{\text{Element type}}, \underbrace{\nu}_{\text{Number of rows}}, \underbrace{\nu}_{\text{Number of columns}})$$

with ν being the value type. The value type can either be a value variable or a concrete value.

The matrix size inference is incorporated into the HM type inference by adapting the `unify` function. Whenever we encounter a matrix type during the unification process, we call a `unifyValue` function on the two row and column values. The `unifyValue` function works similarly to the `unify` function. First, the function resolves the value expression, thus substituting value variables with their assigned values. Then, if at least one of the resolved value expressions is still a value variable, then the union is constructed and the corresponding value variable dictionary entry is updated. If both resolved expressions are equal, then this value is returned. Otherwise an error is thrown.

2.2 Intermediate Representation

After parsing and typing of the source code is done, it is translated into an intermediate representation. The additional abstraction layer allows Gilbert to apply language independent optimization. Moreover, the higher-level abstraction of mathematical operations holds more potential for algebraic optimization. The intermediate format consists of a set of operators to represent the different linear algebra operations. Every operator has a distinct result type and a set of parameters which are required as inputs. The set of intermediate operators can be distinguished into three categories: *Creation operators*, *transformation operators* and *output operators*.

Creation Operators. Creation operators generate or load some data. The `load` operator loads a matrix from disk with the given number of rows and columns. The `eye` operator generates an identity matrix of the requested size. `zeros` generates a matrix of the given size which is initialized with zeros. `Randn` takes the size of the resulting matrix and the mean and the standard deviation of a Gauss distribution. The Gauss distribution is then used to initialize the matrix randomly.

Transformation Operators. The transformation operators constitute the main abstraction of the linear algebra operations. They group operations with similar properties and thus allow an easier reasoning and optimization of the underlying program. The `UnaryScalar-Transformation` takes a single scalar value and applies an unary operation on it. The `ScalarScalarTransformation` constitutes a binary operation on scalar values whereas `ScalarMatrixTransformation` and `MatrixScalarTransformation` represent a binary operation between a scalar and a matrix value. The `VectorwiseMatrixTransformation` applies an operation on each row vector of the given matrix. A vectorwise operation produces a scalar value for each row vector. The `AggregateMatrixTransformation` applies an operation to all matrix entries producing a single scalar result value. The iteration mechanism is represented by the `FixpointIterationMatrix` and `FixpointIterationCellArray` operators.

Writing Operators. The writing operators are used to write the computed results back to disk. There exists a writing operation for each supported data type: `WriteMatrix`, `WriteScalar`, `WriteString`, `WriteCellArray` and `WriteFunction`.

2.3 Gilbert Optimizer

The Gilbert optimizer applies algebraic optimizations to a Gilbert program. The optimizer exploits equivalence transformations which result from the commutative and associative properties of linear algebra operations. It works on the intermediate format of a program, which provides an appropriate high-level abstraction.

Matrix Multiplication Reordering. Matrix multiplications belong to the most expensive linear algebra operations in terms of computational as well as space complexity. Intermediate results of successive matrix multiplications can easily exceed the memory capacity and thus rendering its computation infeasible. However, a smart execution order can sometimes avoid the materialization of excessively large matrices. The best execution order of successive multiplications is the one that minimizes the maximum size of intermediate results. In order to determine the best execution order, the optimizer first extracts all matrix multiplications with more than 2 operands. Then, it calculates the maximum size of all occurring intermediate results for each evaluation order. In order to do this calculation, the optimizer relies on the operands' automatically inferred matrix sizes. At last, it picks the execution order with the minimal maximum intermediate matrix size.

Transpose Pushdown. Transpose pushdown tries to move the transpose operations as close to the matrix input as possible. Thereby, consecutive transpose operations accumulate at the inputs and unnecessary operations erase themselves. Consider, for example, $C = A^T B$ and $E = (CD^T)^T$. By inserting C into E, the expression $E = (A^T BD^T)^T$ is obtained which is equivalent to $DB^T A$. The latter formulation contains only one transpose operation. Usually multiple transpose operations occur because they are written for convenience reasons at various places in the code. Moreover, in complex programs it is possible that the programmer loses track of them or simply is unaware of the optimization potential. Therefore, transpose pushdown can be a beneficial optimization.

3 Gilbert Runtime

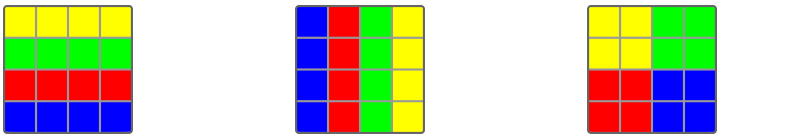
The Gilbert runtime is responsible for executing the compiled MATLAB code on a particular platform. For this purpose, it receives the intermediate representation of the code and translates it into the execution engine's specific format. Currently, Gilbert supports three different execution engines: *LocalExecutor*, *FlinkExecutor* and *SparkExecutor*. The *LocalExecutor* executes the compiled MATLAB code locally. For the distributed execution Gilbert supports Apache Spark and Apache Flink as backends.

The *LocalExecutor* is an interpreter for the intermediate code. It takes the dependency tree of a MATLAB program and executes it by evaluating the operators bottom-up. In order to evaluate an operator, the system first evaluates all inputs of an operator and then executes the actual operator logic. Since the program is executed locally, the complete data of each operator is always accessible and, thus, no communication logic is required. All input and output files are directly written to the local hard drive.

In contrast to the *LocalExecutor*, the *FlinkExecutor* and *SparkExecutor* execute the program in parallel. Consequently, data structures are needed which can be distributed across several nodes and represent the commonly used linear algebra abstractions, such as vectors and matrices. Moreover, the linear algebra operations have to be adjusted so that they keep working in a distributed environment. Since both systems offer a similar set of programming primitives, they can operate on the same data structures. Furthermore, most of the linear algebra operations are implemented in a similar fashion. The details of the distributed data structures and operations are explained in Sect. 3.1 and Sect. 3.2.

3.1 Distributed Matrix Representation

An important aspect of distributed algorithms is their data partitioning. Since the distribution pattern directly influences the algorithms, one cannot consider them independently from one another. In Gilbert's use case, the main data structure are matrices. Thus, the matrix entries have to be partitioned. A first idea could be to partition a matrix according to their rows or columns, as it is depicted in Fig. 2a and Fig. 2b. This scheme allows to represent a matrix as a distributed set of vectors. Furthermore, it allows an efficient realization of cellwise operations, such as $+$, $-$, $/$ or $.*$. In order to calculate the result of such an operation, we only have to join the corresponding rows of both operands and execute the operation locally for each pair of rows.



a Row partitioning b Column partitioning c Square block partitioning

Fig. 2: Row-wise, column-wise and square block-wise matrix partitioning.

This approach, however, unveils its drawbacks when multiplying two equally partitioned matrices A and B. In such a case, the row r of A and the complete matrix B is needed to calculate the resulting row with index r. This circumstance implies a complete repartitioning of B. The repartitioning is especially grave, since B has to be broadcasted to every row of A. In order to quantify the repartitioning costs of such a matrix multiplication, a simple cost model is developed. First of all, it is limited to modeling the communication costs, since network I/O usually constitutes the bottleneck of distributed systems. For the sake of simplicity, the multiplication of two quadratic matrices $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times n}$ giving the matrix $C \in \mathbb{R}^{n \times n}$ is considered. Moreover, we assume that there are n worker nodes available, each of which receiving a single row of A and B, respectively. Thus, A and B are row-wise partitioned. We further assume that rows with the same index are kept on the same worker node. Each row a_r requires the complete knowledge of matrix B in order to produce the row c_r. Therefore, every row b_r has to be sent to all other worker nodes. Thus, the number of rows sent by each worker node is $n - 1$. All of the n worker nodes have to do the same. Consequently, the total number of sent messages is $n(n-1)$ and each message has a size of $n\alpha$ where α is the size of a matrix entry. Usually, each sending operation causes some constant overhead inflicted by resource allocation. This overhead is denoted by Δ. Since all sending operations occur in parallel, the costs caused by constant overhead are

$(n-1)\Delta$. The total amount of data, which has to be sent over the network, is $n^2(n-1)\alpha$. The network interconnection is assumed to guarantee every node a bandwidth ν. Therefore, the time needed for sending the data is $\frac{n^2(n-1)\alpha}{\nu}$. These considerations lead to the following repartitioning cost model:

$$cost_{row} = \frac{n^2(n-1)\alpha}{\nu} + (n-1)\Delta$$

Row and column partitioning are extreme forms of blocking. A less extreme form is to split the matrix into equally sized quadratic blocks as shown in Fig. 2c. In order to identify the individual blocks, each of them will get a block row and block column index assigned. Thus, blocking adds some memory overhead in the form of index information. The blocks are distributed across the worker nodes. The block size directly controls the granularity of the partitioning. Increasing the block size will reduce the memory overhead of distributed matrices while reducing the degree of parallelism. Thus, the user has to adjust the block size value depending on the matrix sizes and the number of available worker nodes in order to obtain best performance. The quadratic block partitioning has similar properties like the row- and column-wise partitioning scheme when it comes to cellwise operations. We simply have to join corresponding blocks with respect to the pair of block row and column index and execute the operation on matching blocks locally. But how does this pattern performs for matrix multiplications? The assumptions are the same as before and additionally it is assumed that n is a square number. Since the matrices A, B and C are equally partitioned into square blocks, indices will henceforth reference the block and not the matrix entry. In order to calculate the block c_{ij}, we have to know the block row a_i and the block column b_j. The resulting block will be stored on the node n_{ij} which already contains the blocks a_{ij} and b_{ij}. Thus, each node n_{ij} has to receive the missing $2\left(\sqrt{n}-1\right)$ blocks from the block row a_i and block column b_j. In total, all worker nodes have to send $2n\left(\sqrt{n}-1\right)$ blocks. Each block has the size $n\alpha$. The total communication costs comprises the transmission costs and the network overhead:

$$cost_{squareBlock} = \frac{2n^2\left(\sqrt{n}-1\right)\alpha}{\nu} + 2\left(\sqrt{n}-1\right)\Delta$$

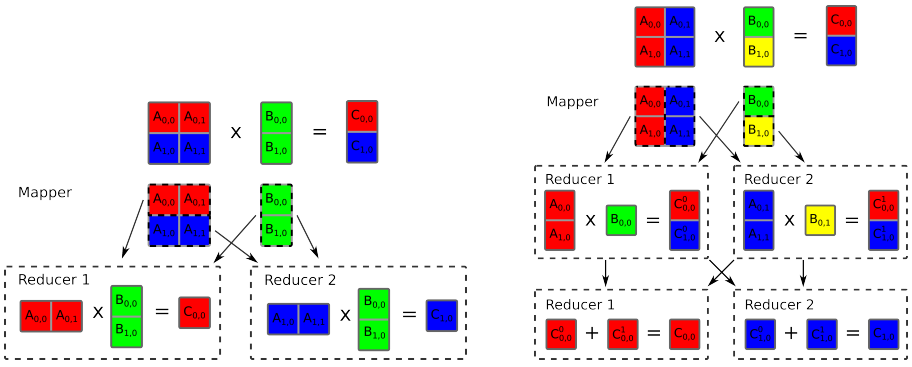
We see that the term $(n-1)$ is replaced by $2\left(\sqrt{n}-1\right)$ in the square block cost model. For $n > 2$, the square block partitioning scheme is thus superior to the row- and column-wise partitioning pattern with respect to the cost model. The reason for this outcome is that the square blocks promote more localized computations compared to the other partitionings. Instead of having to know the complete matrix B, we only have to know one block row of A and one block column of B to compute the final result.

Due to these advantages and its simplicity, we decide to implement the square block partitioning scheme in Gilbert. It would also be possible to combine different partitionings and select them dynamically based on the data sizes and input partitionings. Besides the partitioning, Gilbert also has to represent the individual matrix blocks. There exist several storing schemes for matrices depending on the sparsity of the matrix. For example, if a matrix is dense, meaning that it does not contain many zero elements, the elements are best stored in a continuous array. If a matrix is sparse, then a compressed representation is best suited. Gilbert chooses the representation for each block dynamically. Depending on the non-zero elements to total elements ratio, a sparse or dense representation is selected.

3.2 Linear Algebra Operations

In the following, we will outline the implementation of the matrix multiplication operator, which is most critical for performance in linear algebra programs. For a reference of the implementation of the remaining operators, we refer the interested reader to [Ro14]. Note that we use the Breeze [09] library for conducting local linear algebraic operations. In a MapReduce-like system there exist two distinct matrix multiplication implementations for square blocking. The first approach is based on replicating rows and columns of the operands and is called replication based matrix multiplication (RMM). The other method is derived from the outer product formulation of matrix multiplications. It is called cross product based matrix multiplication (CPMM).

Let us assume that we want to calculate $A \times B = C$ with A, B and C being matrices. The block size has been chosen such that A is partitioned into $m \times l$ blocks, B is partitioned into $l \times n$ blocks and the result matrix C will be partitioned into $m \times n$ blocks. In order to reference the block in the ith block row and jth block column of A, we will write A_{ij}. A block row will be denoted by a single subscript index and a block column by a single superscript index. For example, A_i marks the ith block row of A and A^j the jth block column of A. The replication-based strategy will copy for each C_{ij} the ith block row of A and the jth block column of B. The replicated blocks of A_i and B^j will be grouped together. These steps can be achieved by a single mapper. Once this grouping is done, the final result C_{ij} can be calculated with a single reducer. The reduce function simply has to calculate the scalar product of A_i and B^j. It is important to stress that A_i is replicated for each $C_{ik}, \forall k$. The whole process is illustrated in Fig. 3a.



a Replication based matrix multiplication. b Cross product matrix multiplication.

Fig. 3: Execution strategies for matrix multiplication in MapReduce.

In contrast to RMM, CPMM calculates the outer products between A^k and B_k for all k. A mapper can group the A^k and B_k together so that a reducer can compute the outer products. Consequently, this method does not replicate any data. The outer product produces intermediate result matrices C^k which have to be added up to produce the final result $C = \sum_{k=1}^{l} C^k$. This summation can be achieved by a subsequent reducer. The whole process is illustrated in Fig. 3b.

The methods RMM and CPMM differ in terms of network communication. The former method can be realized within a single MapReduce job whereas the latter requires two. Neither RMM nor CPMM is always superior. The optimal matrix multiplication strategy depends on the matrix size of its operands A and B. Fortunately, Flink and Spark exhibit a little bit more flexibility in terms of higher order functions. Looking at the definition of the matrix multiplication for $C_{ij} = \sum_{k=1}^{l} A_{ik} \times B_{kj}$, it can be seen that every A_{ik} has to be joined with its corresponding $B_{kj}, \forall k$. This pairwise mapping can be easily achieved by using the join function. The join-key is the column index of A and the row index of B. The joiner calculates for each matching pair A_{ik} and B_{kj} an intermediate result C_{ij}^{k}. Grouping the intermediate results with respect to the index pair (i, j) allows us to compute the final result in a subsequent reduce step. The overall algorithm strongly resembles the CPMM.

Flink supports several execution strategies for the higher-order functions. A cost-based optimizer selects the best strategies prior to execution. One possible optimization concerns the join function. The join can either be realized using a broadcast hash join, a repartitioning hash join or a repartitioning sort-merge join algorithm depending on the current partitioning and the input data sizes. If one input data is relatively small compared to the other input, it is usually more efficient to use the broadcast hash join algorithm.

Without loss of generality, we assume that the matrix B constitutes such a small input. If we further assume that the block rows of A are kept on the same worker node, then the last reduce operation can be executed locally and without any shuffling. The resulting execution plan under these assumptions is equivalent to the RMM. If the system chooses any of the repartitioning join algorithms instead, then the columns of A will be distributed across the worker nodes. Consequently, the last reduce step causes a mandatory repartitioning. Then, the resulting execution plan is equivalent to the CPMM. Even though Gilbert has only one dataflow plan specified to implement the matrix multiplication, the Flink system can choose internally between the RMM and CPMM strategy. The strategy is selected by the optimizer which bases its decision on the data size and the partitioning, if available.

4 Evaluation

In this chapter, we will investigate the scalability of Gilbert and its performance compared to well known hand-tuned ML algorithms. We show that Gilbert is not only easily usable in terms of programming but also produces results with decent performance.

Experimental Setup. For our evaluation, we use a Google compute engine cluster with 8 machines of type `n1-standard-16`. Each machine is equipped with 60 gigabytes of main memory and has 16 virtual CPUs. The Spark execution engine runs on Apache Spark-2.0.0 [14]. For the Flink execution engine, we use Apache Flink-1.1.2 [16]. As underlying distributed file system we use Apache Hadoop-2.7.3 [08].

4.1 Scalability

The scalability evaluation investigates how Gilbert behaves under increasing work loads and how well it can exploit varying cluster sizes. As we have implemented Gilbert to provide a scalable linear algebra environment, it is important that it can process data sizes exceeding the main memory of a single machine.

Matrix Multiplication. As a first benchmark, we choose the matrix multiplication $A \times B$ with $A, B \in \mathbb{R}^{n \times n}$ and n being the dimensionality. The matrix multiplication operation is demanding, both in CPU load as well as network I/O as it requires two network shuffles. The matrices A and B are sparse matrices with uniformly distributed non-zero cells. They are randomly generated prior to the matrix multiplication with a sparsity of 0.001. As baseline, we run the same matrix multiplication on a single machine of the cluster using Gilbert's local execution engine. The Flink and Spark execution engines are both started with 48 gigabytes of main memory for their task managers and they distribute the work equally among all worker nodes. In the first experiment, we fixed the block sizes to 500×500 and set the number of cores to 64. We then increased the dimensionality n of A and B to observe the runtime behavior. The resulting execution times for the local, Flink and Spark execution engines are shown in Fig. 4a. In the second experiment, we investigate the scaling behavior of Gilbert with respect to the cluster size. In order to observe the communication costs, we keep the work load per core constant while increasing the number of cores. For this experiment we vary the number of cores from 1 to 128 and scale n such that $n^3/\#cores$ is constant. We started with $n = 5000$ for a single core and reached $n = 25000$ on 128 cores. The results of the experiment are shown in Fig. 4b.

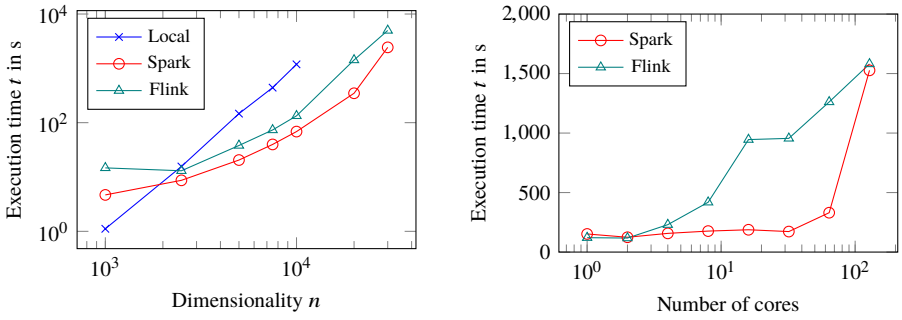


Fig. 4: Scalability of matrix multiplication. a Execution time of matrix multiplication depending on the data size. b Execution time of matrix multiplication depending on the cluster size with constant work load per core.

On a single machine, we are able to execute the matrix multiplication for dimensionalities up to $n = 10000$ in a reasonable time. We can see that the local execution performs better for dimensionalities $n \leq 2500$. That is expected since the matrix still fits completely in the main memory of a single machine and the distributed execution adds communication overhead and job start up latency. For matrices with $n > 2500$, Spark and Flink start to

calculate the matrix multiplication faster than the local executor. We also see that Gilbert can handle matrix sizes which scale far beyond what a single machine can handle.

The results depicted in Fig. 4b indicate for both execution engines decent scale-out behavior. For Spark we observe an almost horizontal line for number of cores ≤ 32. For larger number of cores, the scaling degrades which is probably caused by spilling. The same applies to Flink. However, we can observe that the system has to start spilling data earlier. Thus, Flink's scale-out behavior is not as good as Spark's with respect to matrix multiplication.

Gaussian Non-negative Matrix Factorization. As second benchmark, we choose the Gaussian non-negative matrix factorization (GNMF) algorithm [SL01]. GNMF finds for a given matrix $V \in \mathbb{R}^{d \times w}$ a factorization $W \in \mathbb{R}^{d \times t}$ and $H \in \mathbb{R}^{t \times w}$ such that $V \approx WH$ holds true. For our evaluation, we calculate one step of the GNMF algorithm. We set $t = 10$, $w = 100000$ and vary the number of rows d of V. The matrix $V \in \mathbb{R}^{d \times 100000}$ is a sparse matrix whose sparsity is set to 0.001. The non-zero cells of V are uniformly distributed. As baseline, we run the GNMF on a single machine of the cluster using the local execution engine. Like for the matrix multiplication benchmark, the task manager of Spark and Flink are started with 48 gigabytes of memory. In the first experiment we fix the number of cores to 64. We start with $d = 500$ and increase the number of rows of V to 150000. The block size of Gilbert is set to 500×500. In order to compare the results with the optimized GNMF MapReduce implementation proposed in [Li10], we re-implemented the algorithm using Spark and Flink. This hand-tuned implementation, henceforth denoted as HT-GNMF (HT in the graphs), is also executed on 64 cores. The execution times of HT-GNMF and Gilbert's GNMF are shown in Fig. 5a. In the second experiment of the GNMF benchmark, we analyze how Gilbert scales-out when increasing the cluster size while keeping the work load for each core constant. We vary the cluster size from 1 core to 64 cores and scale the number of documents d accordingly. Initially we start with 1000 documents and, consequently, calculate the matrix factorization for 64000 documents on 64 cores. The results of this experiment are shown in Fig. 5c.

The local executor is applied to sizes of d ranging from 500 to 3000 rows. Surprisingly, the local execution is outperformed by the distributed engines. This fact indicates that the network communication costs are not decisive. The distributed systems can also be used for data sizes which can no longer be handled by a single machine. Both distributed execution engines scale well and achieve almost identical results as the hand tuned implementations. The speedup of the HT-GNMF variants compared to GNMF on Spark's and Flink's executor is shown in Fig. 5b. In the case of Flink, the HT-GNMF variant runs at most 1.78 times faster than Gilbert's GNMF implementation for large data sets. For Spark, we can only observe a maximum speedup of 1.35. This result underlines Gilbert's good performance.

Furthermore, the development with Gilbert was considerably easier. One GNMF step can be programmed in five lines of Gilbert code, whereas we needed 28 lines of Scala code for Spark's HT-GNMF and 70 lines of Scala code for Flink's HT-GNMF. Not only did we have to know how to program Spark and Flink, but it also took us quite some time to verify the correct functioning of both implementations. The verification was made significantly more difficult and time-consuming due to a programming bug we introduced. The debugging process showed us quite plainly how tedious the development process even

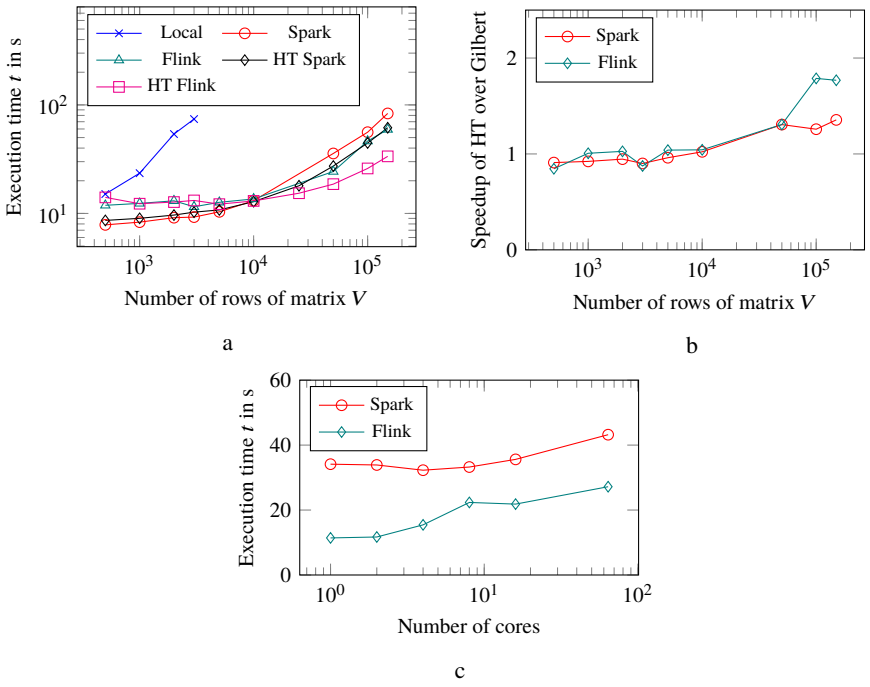


Fig. 5: Scalability of GNMF. a Execution time of one GNMF step depending on the data size. b Speedup of HT-GNMF over Gilbert's GNMF using Spark and Flink. c Execution time of one GNMF step depending on the cluster size with constant work load per core.

with systems like Spark and Flink can be. Thus, the productivity increase gained by using a high-level declarative programming language for linear algebra must not be neglected and compensates for the slight performance loss. [Al10] made similar observations while developing a declarative programming language for distributed systems programming. The scale-out behavior of the Flink and Spark execution engines Fig. 5c both show good results for $\#cores$ up to 64. The runtime on Spark with 64 cores is only 1.26 times slower than the runtime on a single core with the same work load per core. For Flink we observe a slowdown of 2.38 when running on 64 cores.

4.2 PageRank

In this section, we want to investigate how well the PageRank algorithm [Pa98] implemented in Gilbert performs compared to a specialized implementation (denoted as SP in the figures). We expect that the Gilbert runtime adds some overhead. Furthermore, the high-level linear algebra abstraction of Gilbert might make it difficult to exploit certain properties to speed up the processing. We first execute PageRank directly in Gilbert, given the Gilbert code in List. 3, and then run them directly on Flink and Spark. In contrast to Gilbert, the direct implementation requires a deep understanding of the underlying runtime system.

Furthermore, the distributed implementation is far from trivial compared to the linear algebra representation of the original problem.

```
% load adjacency matrix
A = load();
maxIterations = 10;
d = sum(A, 2); % outdegree per vertex
% create the column-stochastic transition matrix
T = (diag(1 ./ d) * A)';
r_0 = ones(numVertices, 1) / numVertices; % initialize the ranks
e = ones(numVertices, 1) / numVertices;
% PageRank calculation
fixpoint(r_0, @(r) .85 * T * r + .15 * e, maxIterations)
```

Listing 3: Gilbert PageRank implementation.

The specialized Spark and Flink implementation of PageRank works as follows. The PageRank vector is represented by a set of tuples (w_i, r_i) with w_i denoting the web site i and r_i being its rank. The adjacency matrix is stored row-wise as a set of tuples (w_i, A_i) with w_i denoting the web site i and A_i being its adjacency list. For the adjacency list A_i, it holds that $w_j \in A_i$ if and only if there exists a link from web site i to j. The next PageRank vector is calculated as follows. The PageRank r_i of web site i is joined with its adjacency list A_i. For each outgoing link $w_j \in A_i$ a new tuple $(w_j, 0.85 r_i / |A_i|)$ with the rank of i being divided by the number of outgoing links is created. In order to incorporate the random jump behavior to any available web site, a tuple $(w_i, 0.15/|w|)$, with $|w|$ being the number of all web sites, is generated for each web site i. In order to compute the next PageRank vector, all newly generated tuples are grouped according to their ID and summed up. These steps constitute a single PageRank iteration whose data flow plan is depicted in Fig. 6.

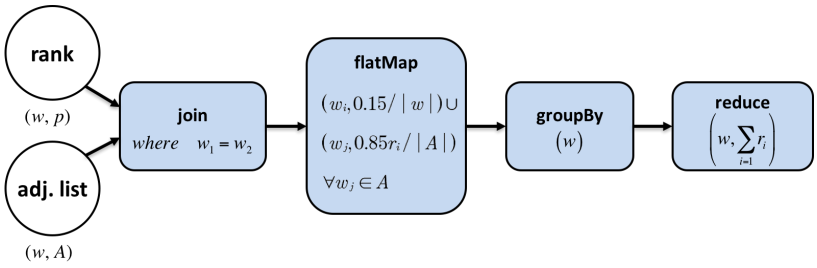


Fig. 6: Data flow of one iteration of the PageRank algorithm for Spark and Flink.

Experiment. For comparison, 10 steps of the PageRank algorithm for varying sizes of the adjacency matrix A are calculated. The randomly generated adjacency matrix A is a sparse matrix of size $n \times n$ with a sparsity of 0.001. The computation is executed on 64 cores with a block size of 500×500. The execution times are depicted in Fig. 7a.

The graphs in Fig. 7a show that the specialized PageRank algorithm runs faster than Gilbert's versions of PageRank. Furthermore, the specialized algorithms show a better scalability.

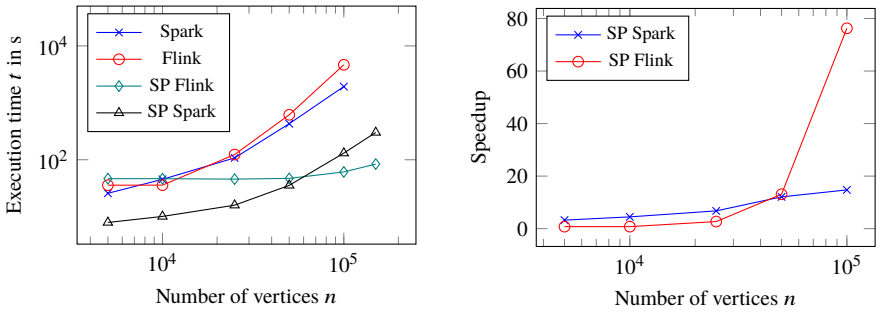


Fig. 7: Comparison of Gilbert's PageRank implementation with specialized algorithms on Spark and Flink. a Execution time of 10 steps of the PageRank algorithm depending on the adjacency matrix's size. b Speedup of specialized algorithms with respect to Gilbert's implementations.

The speedup of the specialized algorithms with respect to Gilbert's implementations for this experiment is shown in Fig. 7b. For $n \leq 50000$, the hand-tuned algorithms running on Spark and Flink show a similar speedup. The Flink and Spark version achieve a speedup of approximately 13 and 12 for $n = 50000$, respectively. However, for $n = 100000$ we can observe a speedup of 76 for the Flink specialized implementation whereas Spark reaches a speedup of 14. Gilbert's performance degradation with Flink is likely caused by earlier data spilling.

The performance difference between the specialized algorithm and Gilbert's version can be explained by considering the different execution plans. As shown in Fig. 6, each iteration of the specialized PageRank algorithm comprises one join, one group reduce and one flat map operation. In contrast, each iteration of Gilbert's PageRank implementation requires two cross, two join and one reduce operation. Thus, it can be clearly seen that Gilbert's high-level linear algebra representation adds three additional operations, with one of them being highly expensive. Therefore, it is not surprising that the specialized PageRank algorithm performs better.

5 Related Work

SystemML [Bo14a; Bo14b; El16; Gh11; Sc15] aims to make machine learning algorithms run on massive datasets without burdening the user with low-level implementation details and tedious hand-tuning. Therefore, it provides an R-like declarative high-level language, called Declarative Machine learning Language (*DML*), and compiles and optimizes the resulting programs to distributed dataflow systems. The linear algebra operations are translated into a directed acyclic graph of high-level operators. This abstract representation allows to apply several optimizations such as algebraic rewrites, choice of internal matrix representation and cost-based selection of the best execution plan. Afterwards these plans are translated to low-level operators and executed either in the driver memory or in parallel. SystemML is one of the main inspirations for Gilbert. While it originally only supported MapReduce as

runtime, it has recently also moved to supporting more advanced dataflow systems. Gilbert differs from SystemML by its fixpoint operator and by leveraging the general optimizers of the underlying system (e.g., the Flink optimizer for matrix multiplication optimization, see Section 3.2).

Samsara [LP16; Sc16] is a domain-specific language for declarative machine learning in cluster environments. Samsara allows its users to specify programs using a set of common matrix abstractions and linear algebraic operations, similar to R or MATLAB. Samsara automatically compiles, optimizes and executes these programs on distributed dataflow systems. With Samsara mathematicians and data scientists can leverage the scalability of distributed dataflow systems via common declarative abstractions, without the need for deep knowledge about the programming and execution models of such systems. Samsara is part of the Apache Mahout library [11] and supports a variety of backends, such as Apache Spark or Apache Flink.

Ricardo [Da10] uses existing technologies to implement a scalable system for statistical analysis: it executes the data shipment via Hadoop and runs the actual analytic computation by R. It integrates these systems via Jaql [Be11], a declarative high-level language for data-processing on Hadoop. Thereby, the user can initiate the distribution, transformation and aggregation of data within Hadoop. Furthermore, the system supports to run R code on the worker nodes as data transformations. However, in this setting the user still has to explicitly specify the data-flow and the data distribution, which requires a substantial understanding of MapReduce's principles. *RHIPE* [Gu12] also tries to extend R to scale to large data sets using Hadoop. RHIPE follows an approach called divide and recombine. The idea is to split the examined data up into several chunks so that they fit in the memory of a single computer. Next a collection of analytic methods is applied to each chunk without communicating with any other computer. After all chunks are evaluated, the results will be recombined in an aggregation step to create the overall analytic result. However, this strict execution pipeline constrains the analysis process considerably. A system which integrates more seamlessly into the R ecosystem is *pR* (parallel R) [Sa09]. The goal is to let statisticians compute large-scale data without having to learn a new system. pR achieves its goal by providing a set of specialized libraries which offer parallelized versions of different algorithms using MPI. However, MPI does not integrate well with an environment where clusters are shared and usually execute several jobs concurrently. Furthermore, it lacks important properties such as fault-tolerance, elasticity and reliability. Another related system for R is *rmpi* [02]. As this is only a wrapper for the respective MPI calls, it also suffers from the aforementioned problems. Furthermore, there is the SparkR [Ve16] project which aims to integrate Spark's API and SQL processing capabilities into R.

For the other popular numerical computing environment out there, namely MATLAB, also a set of parallelization tools exists. The most popular are the MATLAB Parallel Computing Toolbox [Mab] and MATLAB Distributed Computing Server [Maa], which are developed by MathWorks. The former permits to parallelize MATLAB code on a multi-core computer and the latter scales the parallelized code up to large cluster of computing machines. In combination, they offer a flexible way to specify parallelism in MATLAB code. Besides these tools there are also other projects which try to parallelize MATLAB code. The most

noteworthy candidates are pMatlab [BK07] and MatlabMPI [KA04] which shall be named here for the sake of completeness. Unfortunately, none of these approaches is known to integrate well with the current JVM-based Hadoop ecosystem, which is becoming the main source of data in production deployments. Another promising research direction is the deep embedding of the APIs of dataflow systems in their host language [Al15], where the potential to extend this embedding and the resulting optimizations to linear algebraic operations is currently explored [Ku16].

6 Conclusion

Gilbert addresses the problem of scalable analytics by fusing the assets of high-level linear algebra abstractions with the power of massively parallel dataflow systems. Gilbert is a fully functional linear algebra environment, which is programmed by the widespread MATLAB language. Gilbert programs are executed on massively parallel dataflow systems, which allow to process data exceeding the capacity of a single computer's memory. The system itself comprises the technology stack to parse, type and compile MATLAB code into a format which can be executed in parallel. In order to apply high-level linear algebra optimizations, we conceived an intermediate representation for linear algebra programs. Gilbert's optimizer exploits this representation to remove redundant transpose operations and to determine an optimal matrix multiplication order. The optimized program is translated into a highly optimized execution plan suitable for the execution on a supported engine. Gilbert allows the distributed execution on Spark and Flink.

Our systematical evaluation has shown that Gilbert supports all fundamental linear algebra operations, making it fully operational. Additionally, its loop support allows to implement a wide multitude of machine learning algorithms. Exemplary, we have implemented the PageRank and GNMF algorithm. The code changes required to make these algorithms run in Gilbert are minimal and only concern Gilbert's loop abstraction. Our benchmarks have proven that Gilbert can handle data sizes which no longer can be efficiently processed on a single computer. Moreover, Gilbert showed a promising scale out behavior, making it a suitable candidate for large-scale data processing.

The key contributions of this work include the development of a scalable data analysis tool which can be programmed using the well-known MATLAB language. We have introduced the fixpoint operator which allows to express loops in a recursive fashion. The key characteristic of this abstraction is that it allows an efficient concurrent execution unlike the `for` and `while` loops. Furthermore, we researched how to implement linear algebra operations efficiently in modern parallel data flow systems, such as Spark and Flink. Last but not least, we have shown that Flink's optimizer is able to automatically choose the best execution strategy for matrix-matrix multiplications.

Even though, Gilbert can process vast amounts of data, it turned out that it cannot beat the tested hand-tuned algorithms. This is caused by the overhead entailed by the linear algebra abstraction. The overhead is also the reason for the larger memory foot-print, causing Spark and Flink to spill faster to disk. Gilbert trades some performance off for better usability, which manifests itself in shorter and more expressive programming code.

The fact that Gilbert only supports one hard-wired partitioning scheme, namely square blocks, omits possible optimization potential. Interesting aspects for further research and improvements include adding new optimization strategies. The investigation of different matrix partitioning schemes and its integration into the optimizer which selects the best overall partitioning seems to be very promising. Furthermore, a tighter coupling of Gilbert's optimizer with Flink's optimizer could result in beneficial synergies. Forwarding the inferred matrix sizes to the underlying execution system might help to improve the execution plans.

Acknowledgments. This work has been partially supported through grants by the German Federal Ministry for Education and Research for the project Berlin Big Data Center (funding mark 01IS14013A) and by the German Federal Ministry of Economic Affairs and Energy for the project Smart Data Web (funding mark 1MD15010A).

References

[02] Rmpi - MPI for R, 2002, URL: http://cran.r-project.org/web/packages/Rmpi/index.html, visited on: 10/06/2016.

[08] Apache Hadoop, 2008, URL: http://hadoop.apache.org/, visited on: 10/06/2016.

[09] Scala Breeze, 2009, URL: https://github.com/scalanlp/breeze, visited on: 12/11/2016.

[11] Apache Mahout, 2011, URL: http://mahout.apache.org/, visited on: 10/06/2016.

[14] Apache Spark, 2014, URL: http://spark.apache.org/, visited on: 10/06/2016.

[16] Apache Flink, 2016, URL: http://flink.apache.org/, visited on: 09/16/2016.

[93] The R Project for Statistical Computing, 1993, URL: http://www.r-project.org/, visited on: 10/06/2016.

[Al10] Alvaro, P.; Condie, T.; Conway, N.; Elmeleegy, K.; Hellerstein, J. M.; Sears, R.: Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In: Proceedings of the 5th European conference on Computer systems. ACM, pp. 223–236, 2010.

[Al15] Alexandrov, A.; Kunft, A.; Katsifodimos, A.; Schüler, F.; Thamsen, L.; Kao, O.; Herb, T.; Markl, V.: Implicit Parallelism through Deep Language Embedding. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 47–61, 2015.

[Be11] Beyer, K. S.; Ercegovac, V.; Gemulla, R.; Balmin, A.; Eltabakh, M.; Kanne, C.-C.; Ozcan, F.; Shekita, E. J.: Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In: Proceedings of VLDB Conference. Vol. 4, pp. 1272–1283, 2011.

[BK07] Bliss, N. T.; Kepner, J.: pMATLAB Parallel MATLAB Library. International Journal of High Performance Computing Applications 21/3, pp. 336–359, 2007.

[Bo14a] Boehm, M.; Burdick, D. R.; Evfimievski, A. V.; Reinwald, B.; Reiss, F. R.; Sen, P.; Tatikonda, S.; Tian, Y.: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. IEEE Data Eng. Bull. 37/3, pp. 52–62, 2014.

[Bo14b] Boehm, M.; Tatikonda, S.; Reinwald, B.; Sen, P.; Tian, Y.; Burdick, D. R.; Vaithyanathan, S.: Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. Proceedings of the VLDB Endowment 7/7, pp. 553–564, 2014.

[Da10] Das, S.; Sismanis, Y.; Beyer, K. S.; Gemulla, R.; Haas, P. J.; McPherson, J.: Ricardo: Integrating R and Hadoop. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, pp. 987–998, 2010.

[DG08] Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 51/1, pp. 107–113, 2008.

[DM82] Damas, L.; Milner, R.: Principal Type-Schemes for Functional Programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, pp. 207–212, 1982.

[El16] Elgohary, A.; Boehm, M.; Haas, P. J.; Reiss, F. R.; Reinwald, B.: Compressed Linear Algebra for Large-Scale Machine Learning. Proceedings of the VLDB Endowment 9/12, pp. 960–971, 2016.

[Fu09] Furr, M.; An, J.-h. D.; Foster, J. S.; Hicks, M.: Static Type Inference for Ruby. In: Proceedings of the 2009 ACM symposium on Applied Computing. ACM, pp. 1859–1866, 2009.

[Gh11] Ghoting, A.; Krishnamurthy, R.; Pednault, E.; Reinwald, B.; Sindhwani, V.; Tatikonda, S.; Tian, Y.; Vaithyanathan, S.: SystemML: Declarative Machine Learning on MapReduce. In: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering. IEEE, pp. 231–242, 2011.

[Gu12] Guha, S.; Hafen, R.; Rounds, J.; Xia, J.; Li, J.; Xi, B.; Cleveland, W. S.: Large Complex Data: Divide and Recombine (D&R) with RHIPE. Stat 1/1, pp. 53–67, 2012.

[Hi69] Hindley, R.: The Principal Type-Scheme of an Object in Combinatory Logic. Transactions of the American Mathematical Society 146/, pp. 29–60, 1969.

[KA04] Kepner, J.; Ahalt, S.: MatlabMPI. Journal of Parallel and Distributed Computing 64/8, pp. 997–1005, 2004.

[Ku16] Kunft, A.; Alexandrov, A.; Katsifodimos, A.; Markl, V.: Bridging the Gap: Towards Optimization Across Linear and Relational Algebra. In: Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. BeyondMR '16, ACM, 1:1–1:4, 2016.

[Li10] Liu, C.; Yang, H.-c.; Fan, J.; He, L.-W.; Wang, Y.-M.: Distributed Nonnegative Matrix Factorization for Web-Scale Dyadic Data Analysis on MapReduce. In: Proceedings of the 19th international conference on World wide web. WWW '10, ACM, pp. 681–690, 2010.

[LP16] Lyubimov, D.; Palumbo, A.: Apache Mahout: Beyond MapReduce. CreateSpace Independent Publishing Platform, 2016.

[Maa] MathWorks: Matlab Distributed Computing Server, URL: http://www.mathworks.com/products/distriben/, visited on: 10/06/2016.

[Mab] MathWorks: Matlab Parallel Computing Toolbox, URL: http://www.mathworks.de/products/parallel-computing/, visited on: 10/06/2016.

[Ma84] MathWorks: Matlab - The Language for Technical Computing, 1984, URL: http://www.mathworks.com/products/matlab/, visited on: 10/06/2016.

[Mi78] Milner, R.: A Theory of Type Polymorphism in Programming. Journal of computer and system sciences 17/3, pp. 348–375, 1978.

[Pa98] Page, L.; Brin, S.; Motwani, R.; Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. In: Proceedings of the 7th International World Wide Web Conference. Pp. 161–172, 1998.

[Ro14] Rohrmann, T.: Gilbert: A Distributed Sparse Linear Algebra Environment Executed in Massively Parallel Dataflow Systems, MA thesis, Technische Universität Berlin, 2014.

[Sa09] Samatova, N. F.: pR: Introduction to Parallel R for Statistical Computing. In: CScADS Scientific Data and Analytics for Petascale Computing Workshop. Pp. 505–509, 2009.

[Sc15] Schelter, S.; Soto, J.; Markl, V.; Burdick, D.; Reinwald, B.; Evfimievski, A.: Efficient Sample Generation for Scalable Meta Learning. In: 2015 IEEE 31st International Conference on Data Engineering. IEEE, pp. 1191–1202, 2015.

[Sc16] Schelter, S.; Palumbo, A.; Quinn, S.; Marthi, S.; Musselman, A.: Samsara: Declarative Machine Learning on Distributed Dataflow Systems. Machine Learning Systems workshop at NIPS/, 2016.

[SL01] Seung, D.; Lee, L.: Algorithms for Non-Negative Matrix Factorization. Advances in neural information processing systems 13/, pp. 556–562, 2001.

[Ve16] Venkataraman, S.; Yang, Z.; Liu, D.; Liang, E.; Falaki, H.; Meng, X.; Xin, R.; Ghodsi, A.; Franklin, M.; Stoica, I.; Zaharia, M.: SparkR: Scaling R Programs with Spark. In: Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data. SIGMOD '16, ACM, pp. 1099–1104, 2016.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 289

Benchmarking Univariate Time Series Classifiers

Patrick Schäfer[1] Ulf Leser[2]

Abstract: Time series are a collection of values sequentially recorded over time. Nowadays, sensors for recording time series are omnipresent as RFID chips, wearables, smart homes, or event-based systems. Time series classification aims at predicting a class label for a time series whose label is unknown. Therefore, a classifier has to train a model using labeled samples. Classification time is a key challenge given new applications like event-based monitoring, real-time decision or streaming systems. This paper is the first benchmark that compares 12 state of the art time series classifiers based on prediction and classification times. We observed that most of the state-of-the-art classifiers require extensive train and classification times, and might not be applicable for these new applications.

Keywords: Benchmark, Time Series, Classification, Scalability.

1 Introduction

Time series (TS) are a collection of values sequentially recorded over time. TS are increasingly popular due to the growing importance of automatic sensors producing an every increasing flood of large, high-resolution TS in areas such as RFID chips, wearable sensors (wrist bands, smart phones), smart homes [JZ14], or event-based systems [MZJ13]. TS emerge in many applications, like weather observations, industry automation, mobility tracking, etc.

This study focusses on time series classification (TSC). TSC describes the task of predicting a class label for a TS whose label is unknown. Therefore, a classifier has to train a model using labeled TS. The UCR time series classification and clustering archive [Ch15] is a representative selection of TS datasets. Along with the release of these datasets, the authors published accuracy baselines to make TSC publications comparable. The largest UCR datasets contain a few thousand TS of a few thousand measurements. At the same time real-time decision systems emerge with billions of measurements for thousands of sensors. As a concrete example, seizures in long-term human intra-cranial EEG recordings of epilepsy patients have to be predicted [Pr15]. This dataset contains EEG recordings of 10 minutes each and accounts for more than 50GB with $240000 \times 16 \times 6000$ measurements from 6000 samples and 16 electrodes. As another real-world example, energy consumption profiles from smart plugs deployed in households [JZ14] were recorded. This dataset contains four billion measurements from 2125 plugs distributed across 40 households. It aims at load prediction and outlier detection using the power profiles of electrical devices. In such applications a key challenge is to provide scalability in runtime combined with a high classification accuracy.

[1] Humboldt Universität zu Berlin, Germany, patrick.schaefer@hu-berlin.de
[2] Humboldt Universität zu Berlin, Germany, leser@hu-berlin.de

An excellent survey of TSC algorithms is given in [Ba16]. Generally, one can observe a trade-off between accuracy and runtime of TSC algorithms. A trend in TSC is to build ensembles of core classifiers [Ba15, LB14]. While this does increase accuracy significantly, the runtime is negatively affected, as each core classifier has to be trained to build the ensemble and each has to predict a label. Another method is to reduce prediction times at the cost of training times or accuracy. For example, Dynamic Time Warping (DTW) typically considers windows of a certain length only, which both reduces runtime and improves accuracy, if the length limit is set properly [Pe14, LB14]. Similar ideas exist for other techniques such as shapelet classifiers. A runtime optimized version is Fast Shapelets (FS) [RK13].

A recent comparative study [Ba16] compares 18 recent TS classifiers in terms of classification accuracy. Classification accuracy has been the key metric to evaluate new TSC methods [Ba16, Di08, LB14]. This paper presents the first benchmark based on runtimes and accuracy for 12 state-of-the-art TS classifiers. We identified four groups of TS classifiers: whole series, shapelets, bag-of-features, and ensembles. For each group, we have selected the most accurate [Ba16] and fast representatives, if the implementation was available. Combined, our benchmark ran for more than 1000 CPU days. We observed that most of the state-of-the-art classifiers require extensive train and classification times.

The rest of the paper is structured as follows: Section 2 contains the background on TS classification and related work. Section 3 presents state of the art in TSC. An experimental evaluation is presented in Section 4.

2 Background & Related Work

2.1 Definitions

A univariate *time series* is a sequence of $n \epsilon \mathbb{N}$ real values, ordered in time. If the sampling rates of the TS are the same, one can omit the time stamps for simplicity sake and consider the TS as sequences of n-dimensional data points: $T = (t_1, \ldots, t_n), n \epsilon \mathbb{N}$. We associate each TS with a class label $y \epsilon \mathbb{N}$. All TS with the same label represent a class. *Time series classification (TSC)* describes the task of predicting a class label for a TS whose label is unknown.

2.2 UCR time series classification archive

The UCR time series classification and clustering archive [Ch15] is often used as basis for benchmarking and comparing in TS research. It contains a representative sample of univariate TS use cases. In its initial version, it contained 45 datasets. It has recently been expanded to 85 datasets. Each dataset is split into a train and test set. All time series of a dataset have the same length. These datasets include a vast variety of TS, including, motion sensors (inline-skating, gun aiming, cricket), ECG signals, spectrograms, starlight-curves,

and image outlines (anthropology, face or animal contours); 5 of these 85 datasets are synthetic. Some datasets have only a few dozen samples (Beet, Coffee, OliveOil), while the largest contain thousands of TS with thousands of measurements (StarLightCurves, FordA, FordB). In total, there are roughly 50.000 train TS and 100.000 test TS and a total of 55 million measurements.

2.3 Related Work

Previous comparative studies on TS classification had focussed mostly on the accuracy of TS classifiers. In [Di08] 8 TS representations and 9 whole series distance measures on a subset of 38 datasets from the UCR time series archive were benchmarked. They found that there was no whole series distance measure that was superior to others in terms of accuracy. In [LB14] elastic distance measures are compared on 75 TSC problems, with 46 datasets from the UCR archive. They found no significant difference between elastic distance measures and that through ensembling a more accurate classifier than each single core classifier can be created. A recent study [Ba16] compares 18 state-of-the-art classifiers in terms of accuracy, and runtimes have not been evaluated. They implemented all classifiers in a common JAVA framework on 85 UCR datasets. They found that 9 algorithms are significantly more accurate than the baselines (DTW and Rotation Forest). The COTE ensemble [Ba16] was the most accurate in their study. In [Sc16] we proposed a fast classifier and benchmarked its runtime against 5 competitors based on our own implementations or those given by the authors. With the implementation of the state-of-the-art classifiers in [Ba16], we are now in the position to extend our benchmark with 12 state-of-the-art classifiers in terms of accuracy *and runtime*.

3 Approaches to Time Series Classification

TS classifiers can be divided into four groups.

Whole Series: TS are compared by a distance measure applied to the whole TS data. Elastic distance measures compensate for small differences in the TS such as warping in the time axis. They are ill-suited if only TS subsequences are important as in all EEG or ECG signals.

Shapelets: Shapelets are subsequences of a TS that are maximally representative of a class label, and can appear at any offset. A TS can be assigned to a classes by the absence of, presence of, or the Euclidean distance to a shapelet.

Bag-of-Features / Bag-of-Patterns: These approaches distinguish TS by the frequency of occurrence of subsequences rather than their presence of absence. Firstly, subsequences are extracted from TS. Secondly, features are generated from these subsequences, i.e., by generating statistics over or discretization of the subsequences. Finally, these feature vectors are used as input to standard classifiers.

Ensembles: Ensembles combine different core classifiers (shapelets, bag-of-patterns, whole series) into a single classifier. Each core classifier produces a label and a (majority) vote is performed. These classifiers have shown to be highly accurate at the cost of an increased runtime [Ba16].

3.1 Whole Series

Dynamic Time Warping (DTW) [Ra12]: DTW is an elastic similarity measure as opposed to the Euclidean distance (ED). DTW calculates the optimal match between two TS, given some restraints on the amount of allowed displacement of the time axis. This best warping window size is typically learned by cross validation on a training set. This provides warping invariance and essentially is a peak-to-peak and valley-to-valley alignment of two TS. It is likely to fail if there is a variable number of peaks and valleys in two TS. DTW is commonly used as the baseline to compare to [Di08, LB14, Ba16]. Early abandoning techniques and cascading lower bounds have been introduced in [Ra12], which we implemented for our runtime benchmark.

3.2 Shapelet Approaches

Fast Shapelets (FS) [RK13]: Finding shapelets from a set of TS is very time consuming. Subsequences of variable length have to be extracted at each possible offset of a TS and the distance of the subsequences to all other TS is minimized. These subsequences whose distance best separates between classes are used as shapelets. To speed up shapelet discovery, the FS approach makes use of approximation and random projections. Each candidate is discretized and the word count is stored. Then multiple random projections are generated to allow for single character flips in a word. The frequency of a word after projection approximates the occurrences of a subsequence within all TS. The top k words that best separate between classes are mapped back to the TS subsequences. These subsequences represent the nodes of a decision tree. The distance to each subsequence (shapelet) is used as a branching criterion.

Shapelet Transform (ST) [BB15]: The ST separates the shapelet discovery from the classification step. First, the top k shapelets are extracted from the data. Next, the distance of each TS to all k shapelets is computed to form a new feature space. Finally, standard classifiers such as Naive Bayes, C4.5 decision trees, SVMs, Random Forests, Rotation Forests and Bayesian networks are trained with this feature space. Each classifier is assigned a weight based on the train performance with the aim to build an ensemble using a weighted vote for prediction. ST is the most accurate shapelet approach according to an extensive evaluation in [Ba16].

Learning Shapelets (LS) [Gr14]: In LS the shapelets are synthetically generated as part of an optimization problem, as opposed to extracting them from the samples as in ST or FS. The expressive power of this model is much better, as the algorithm can generate smoothed versions of the subsequences or subsequences that do not exist within the data. The shapelets

are initialized using k-means clustering. This method then uses gradient descent and a logistic regression model to jointly learn the weights of the model and the optimal shapelets.

3.3 Bag-of-Features / Bag-of-Patterns Approaches

Time Series Bag of Features (TSBF) [BRT13]: The TSBF approach extracts random subsequences of random lengths from the TS. It then partitions these into shorter intervals and statistical features are extracted. A codebook is generated with these features using a random forest classifier. Finally, a supervised learner is trained with the codebook.

BoP [LKL12]: The BoP model extracts sliding windows from a TS and discretizes these windows to words using Symbolic Aggregate approXimation (SAX) [Li07]. The best window size has to be learned from training. Discretization is performed by calculating mean values over disjoint subsections of a window. Each mean value is then discretized to a symbol using equi-probable intervals. The frequency of words is recorded in a histogram. The Euclidean distance (ED) between two histograms is used for similarity, which represents the difference in word frequencies.

SAX VSM [SM13]: SAX VSM is based on the BoP approach. However, it uses a tf-idf representation of the histograms. A histogram is built for each class, as opposed to each TS in BoP. Words that occur frequently across all classes obtain a low weight, whereas words unique within a single class obtain a high weight. The Cosine similarity between a TS histogram and the tf-idf class vectors is used for similarity. The use of a tf-idf model instead of 1-nearest neighbour (1-NN) search reduces the computational complexity.

BOSS [Sc15]: A recent bag-of-patterns model is Bag-of-SFA-Symbols (BOSS). Sliding windows are extracted and each window is transformed into a word. In contrast to BoP, it makes use of the truncated Fourier transform and discretizes the real and imaginary parts of the Fourier coefficients to symbols. This discretization scheme is called Symbolic Fourier Approximation (SFA) [SH12]. Both, SAX and SFA have a noise reducing effect, BOSS by the use of the first Fourier coefficients (low-pass filter) and SAX by averaging subsections. BOSS uses an asymmetric distance measure in combination with 1-NN search: only those words that occur in the 1-NN TS query are considered, whereas all words that occur exclusively in the TS sample are ignored. To improve performance, multiple window sizes are ensembled to a single classifier. BOSS is the most accurate bag-of-patterns approach according to [Ba16].

BOSS VS [Sc16]: BOSS VS is based on the BOSS approach and trades accuracy for runtime. It builds a tf-idf representation on top of SFA histograms using the Cosine similarity as distance measure. BOSS VS trains an ensemble using $\sqrt{n}$ window sizes at equi-distance.

3.4 Ensembles

Elastic Ensemble (EE PROP) [LB14]: EE PROP is a combination of 11 nearest neighbour whole series classifiers, including DTW CV, DTW, LCSS, ED. A voting scheme weights each classifier according to its train accuracy.

Collective of Transformation Ensembles (COTE) [Ba15]: COTE is based on 35 different core classifiers in time, autocorrelation, power spectrum and shapelet domain. It is composed of the EE (PROP) ensemble and ST classifier. It is the most accurate TSC according to [Ba16].

4 Experiments

Datasets: We evaluated all TS classifiers using the UCR benchmark datasets [Ch15]. Each dataset provides a train and test split.

Classifiers: We evaluated the state-of-the-art TSC: COTE, EE PROP, BOSS, BOSS VS, BoP, SAX VSM, LS, FS, ST, TSBF, 1-NN DTW and 1-NN DTW CV with a warping window constraint.

Implementation: Where possible, we used the implementation given by the authors [BO16, RK13] or the implementations given by [Ba16]. For 1-NN DTW and 1- NN DTW CV we make use of the state-of-the-art lower bounding techniques [Ra12]. Multi-threaded code is available for BOSS and BOSS VS, but we have restricted all codes to use a single core. We used the standard parameters of each classifier in the experiments.

Machine: All experiments ran on a server running openSUSE with a XeonE7-4830 2.20GHz and 512GB RAM, using JAVA JDK x64 1.8.

4.1 Classification Accuracy

Fig. 1 shows a critical difference diagram over the average ranks of the classifiers as introduced in [De06]. The classifiers with the lowest (best) accumulated ranks are to the right of the plot. Our results are similar to the ones previously published in [Ba16], with two exceptions: the BOSS VS classifier was not part of their experiments, and we have used the original train/test splits rather than resampling.

Whole Series: The 1-NN DTW and 1-NN DTW CV are among the worst (highest) ranked classifiers in our evaluation.

Shapelets: While FS is optimized for performance, ST is optimized for accuracy. As such ST shows the second lowest (best) rank and FS has the highest (worst) rank. ST first extracts shapelets and then trains an ensemble of classifiers on top of the shapelet representation. This might be one reason, why it is more accurate than LS, which is based on one standard classifier.

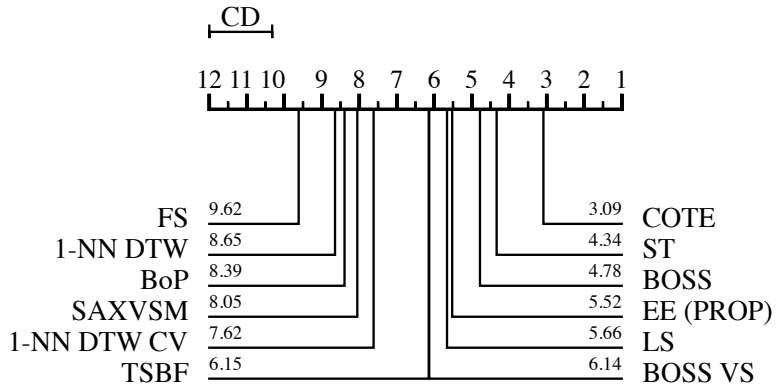


Fig. 1: Average ranks of state-of-the-art classifiers.

Bag-of-Patterns: BOSS and BOP are optimized for accuracy and BOSS VS and SAX VSM are optimized for speed. As such, BOSS shows the highest accuracy of these. Both TSBF and BOSS VS are more accurate than SAX VSM and BoP.

Ensembles: Ensemble classifiers have high accuracy at the cost of computational complexity. They offer a higher classification accuracy than each of the core classifiers, which are part of the ensembles. COTE is the most accurate classifier.

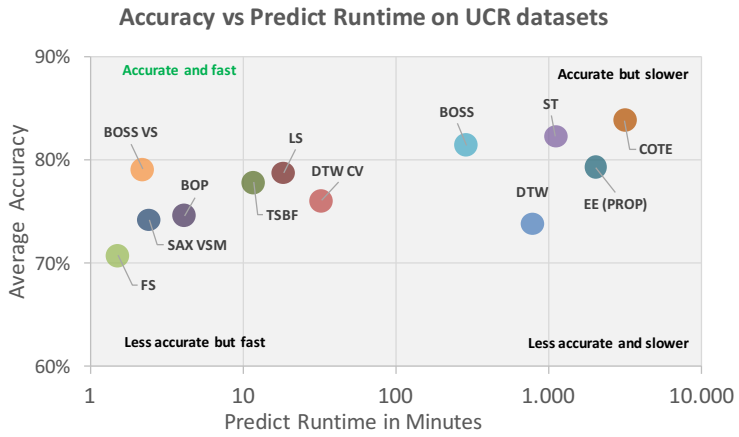
4.2 Runtime

Fig. 2 shows the CPU time on the x-axis (in logarithmic scale) and average accuracy on the y-axis for training (top) and prediction (bottom) of all 12 classifier on the datasets. Our experiments ran for more than 1000 CPU days, thus we had to limit the experiment to the 45 core UCR datasets, because of the high runtime of some classifiers, i.e., ST >1000 hours, EE (PROP) >1800 hours, and COTE >2900 hours for training using default parameters. EE and COTE results are still pending after 6 CPU weeks on the NonInvasiveFatalECGThorax1 and NonInvasiveFatalECGThorax2 datasets. All 45 UCR datasets account for roughly 17000 train and 62000 test TS in total.

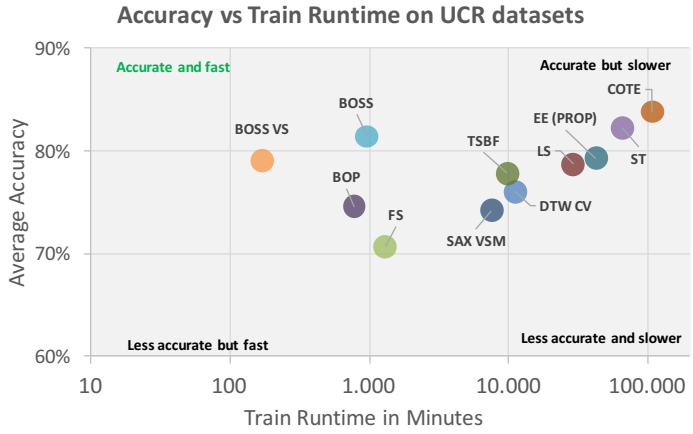
Whole Series: 1-NN DTW CV performs a training step that significantly reduces the runtime for the prediction step. Training DTW CV took 186 hours. DTW CV improves the average accuracy by some percent. Still, DTW CV and DTW show a rather low accuracy.

Shapelets: Shapelets are among the slowest (ST) and fastest classifiers (FS). If accuracy is important, ST is a good choice. If speed is important FS is a better choice. However, FS is the least accurate classifier in our experiments (compare Fig. 1). ST is more than 2 orders of magnitude slower than FS for prediction and training.

Bag-of-Patterns: BOSS VS is a runtime optimized version of BOSS, likewise SAX VSM is an optimized variant of BoP. BOSS VS shows a good trade-off between classification accuracy and runtime. It is orders of magnitude faster than most competitors, and equally



(a) Cumulative prediction (classification) time vs average accuracy. The runtime is in logarithmic scale.



(b) Cumulative train time vs average accuracy. The runtime is in logarithmic scale.

Fig. 2: Runtimes.

fast as FS while offering a much better accuracy. BoP and SAX VSM have rather high train times but fast test times. BOSS has the second highest accuracy and is faster than ST by one to two orders in magnitude.

Ensembles: EE PROP is an ensemble of whole series classifiers. As such it has a higher runtime but offers better accuracy than DTW and DTW CV, which are part of the ensemble. To obtain high accuracy, the COTE ensemble makes use of ST and EE PROP. Thus, its runtime is essentially a composition of these runtimes. Ensembles show the highest test and train runtime in the experiments.

In general, classifiers with high accuracy require time consuming training. By increasing train times, the prediction time can be reduced as for DTW and DTW CV. By ensembling classifiers, accuracy can be increased at the cost of runtime as for COTE, ST or EE (PROP). By sacrificing some accuracy a better runtime can be achieved as for BOSS VS, FS and SAX VSM. The authors of COTE, EE (PROP), and ST emphasize in [Ba16] that their code was not optimized for runtime and that each core classifier can be executed in parallel. However, we consider CPU time, which is independent of the number of cores used.

5 Conclusion

There is a trade off between classification accuracy and computational complexity. To obtain high accuracy, time series classifiers have to perform extensive training. For example the 1-NN DTW classifier can be used with and without a warping window constraint. When the constraint is set, the time for prediction is significantly reduced. However, the time to train the best window size prohibits its application in real-time and streaming contexts. By sacrificing some accuracy, the runtime of a classifier can be reduced by orders of magnitude. Overall, COTE, ST and BOSS show the highest classification accuracy at the cost of increased runtime. BOSS VS offers a good trade off between classification accuracy and runtime, as it is orders of magnitude faster than the most accurate classifiers. However, BOSS VS should be considered a starting point rather than the final solution. Future research in time series classification could lead to producing fast and accurate classifiers.

References

[Ba15] Bagnall, Anthony; Lines, Jason; Hills, Jon; Bostrom, Aaron: Time-Series Classification with COTE: The Collective of Transformation-Based Ensembles. IEEE Transactions on Knowledge and Data Engineering, 27(9):2522–2535, 2015.

[Ba16] Bagnall, Anthony; Lines, Jason; Bostrom, Aaron; Large, James; Keogh, Eamonn: The Great Time Series Classification Bake Off: An Experimental Evaluation of Recently Proposed Algorithms. Extended Version. Data Mining and Knowledge Discovery, pp. 1–55, 2016.

[BB15] Bostrom, Aaron; Bagnall, Anthony: Binary shapelet transform for multiclass time series classification. In: International Conference on Big Data Analytics and Knowledge Discovery. Springer, pp. 257–269, 2015.

[BO16] BOSS implementation: . `https://github.com/patrickzib/SFA/`, 2016.

[BRT13] Baydogan, Mustafa Gokce; Runger, George; Tuv, Eugene: A bag-of-features framework to classify time series. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(11):2796–2802, 2013.

[Ch15] Chen, Y; Keogh, E; Hu, B; Begum, N; Bagnall, A; Mueen, A; Batista, G: , The UCR Time Series Classification Archive. `http://www.cs.ucr.edu/~eamonn/time_series_data`, 2015.

[De06] Demšar, Janez: Statistical comparisons of classifiers over multiple data sets. The Journal of Machine Learning Research, 7:1–30, 2006.

[Di08] Ding, Hui; Trajcevski, Goce; Scheuermann, Peter; Wang, Xiaoyue; Keogh, Eamonn: Querying and mining of time series data: experimental comparison of representationsx and distance measures. 2. VLDB Endowment, pp. 1542–1552, 2008.

[Gr14] Grabocka, Josif; Schilling, Nicolas; Wistuba, Martin; Schmidt-Thieme, Lars: Learning time-series shapelets. In: 2014 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 392–401, 2014.

[JZ14] Jerzak, Zbigniew; Ziekow, Holger: The DEBS 2014 Grand Challenge. In: 2014 ACM International Conference on Distributed Event-based Systems. ACM, pp. 266–269, 2014.

[LB14] Lines, Jason; Bagnall, Anthony: Time series classification with ensembles of elastic distance measures. Data Mining and Knowledge Discovery, 29(3):565–592, 2014.

[Li07] Lin, Jessica; Keogh, Eamonn J.; Wei, Li; Lonardi, Stefano: Experiencing SAX: a novel symbolic representation of time series. Data Mining and knowledge discovery, 15(2):107–144, 2007.

[LKL12] Lin, Jessica; Khade, Rohan; Li, Yuan: Rotation-invariant similarity in time series using bag-of-patterns representation. Journal of Intelligent Information Systems, 39(2):287–315, 2012.

[MZJ13] Mutschler, Christopher; Ziekow, Holger; Jerzak, Zbigniew: The DEBS 2013 grand challenge. In: 2013 ACM International Conference on Distributed Event-based Systems. ACM, pp. 289–294, 2013.

[Pe14] Petitjean, François; Forestier, Germain; Webb, Geoffrey I; Nicholson, Ann E; Chen, Yanping; Keogh, Eamonn: Dynamic Time Warping averaging of time series allows faster and more accurate classification. In: 2014 IEEE International Conference on Data Mining. IEEE, pp. 470–479, 2014.

[Pr15] Predict seizures in long-term human intracranial EEG recordings: . `https://www.kaggle.com/c/melbourne-university-seizure-prediction`, 2015.

[Ra12] Rakthanmanon, Thanawin; Campana, Bilson; Mueen, Abdullah; Batista, Gustavo; Westover, Brandon; Zhu, Qiang; Zakaria, Jesin; Keogh, Eamonn: Searching and mining trillions of time series subsequences under dynamic time warping. In: 2012 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 262–270, 2012.

[RK13] Rakthanmanon, Thanawin; Keogh, Eamonn: Fast Shapelets: A Scalable Algorithm for Discovering Time Series Shapelets. In: 2013 SIAM International Conference on Data Mining. SIAM, 2013.

[Sc15] Schäfer, Patrick: The BOSS is concerned with time series classification in the presence of noise. Data Mining and Knowledge Discovery, 29(6):1505–1530, 2015.

[Sc16] Schäfer, Patrick: Scalable time series classification. Data Mining and Knowledge Discovery, 30(5):1273–1298, 2016.

[SH12] Schäfer, Patrick; Högqvist, Mikael: SFA: a symbolic fourier approximation and index for similarity search in high dimensional datasets. In: 2012 International Conference on Extending Database Technology. ACM, pp. 516–527, 2012.

[SM13] Senin, Pavel; Malinchik, Sergey: SAX-VSM: Interpretable time series classification using SAX and vector space model. In: 2013 IEEE International Conference on Data Mining. IEEE, pp. 1175–1180, 2013.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 299

Incremental ETL Pipeline Scheduling for Near Real-Time Data Warehouses

Weiping Qu[1], Stefan Deßloch[2]

Abstract: We present our work based on an incremental ETL pipeline for on-demand data warehouse maintenance. Pipeline parallelism is exploited to concurrently execute a chain of maintenance jobs, each of which takes a batch of delta tuples extracted from source-local transactions with commit timestamps preceding the arrival time of an incoming warehouse query and calculates final deltas to bring relevant warehouse tables up-to-date. Each pipeline operator runs in a single, non-terminating thread to process one job at a time and re-initializes itself for a new one. However, to continuously perform incremental joins or maintain slowly changing dimension tables (SCD), the same staging tables or dimension tables can be concurrently accessed and updated by distinct pipeline operators which work on different jobs. Inconsistencies can arise without proper thread coordinations. In this paper, we proposed two types of consistency zones for SCD and incremental join to address this problem. Besides, we reviewed existing pipeline scheduling algorithms in our incremental ETL pipeline with consistency zones.

1 Introduction

With increasing demand for real-time analytic results on data warehouses, the frequency of refreshing data warehouse tables is increasing and the time window for executing an ETL (Extract-Transform-Load) job is shrinking (to minutes or seconds). The design of data warehouses and ETL maintenance flows is driven by not only efficiency but also data freshness considerations. For efficiency, incremental ETL techniques [BJ10] have been widely used in near real-time ETL flows and propagate deltas (insertions/deletions/updates) from source tables to target warehouse tables instead of recomputing from scratch. Incremental ETL is similar to materialized view maintenance while one of the differences is that view maintenance jobs are bracketed into internal transactions to make materialized views transactionally consistent with base tables, while ETL flows are generally executed by external tools without full transaction support. The consistency of data warehouses has been addressed in previous work [ZGMW96, TPL08, GJ11] by applying a range of techniques to ETL processes. In our work, the maintenance of warehouse tables is triggered immediately by incoming queries (referred to as *on-demand/lazy/deferred maintenance*). At the time each query arrives, it is suspended first in the system while a maintenance job is constructed and propagates only those source-local transactions that have commit timestamps preceding the query arrival time and have not yet been synchronized with warehouse tables. The query resumes execution when the warehouse tables are brought up-to-date by this maintenance job. An arrival of a sequence of queries forces our ETL flows to work on a sequence of

[1] TU Kaiserslautern, AG Heterogene Informationssysteme, qu@informatik.uni-kl.de
[2] TU Kaiserslautern, AG Heterogene Informationssysteme, dessloch@@informatik.uni-kl.de

maintenance jobs (called *maintenance job chain*), each of which brings relevant warehouse tables to the correct state demanded by a specific query. For efficiency, we exploit pipeline parallelism and proposed an idea of *incremental ETL pipeline* in [Qu15]. Figure 1 shows an

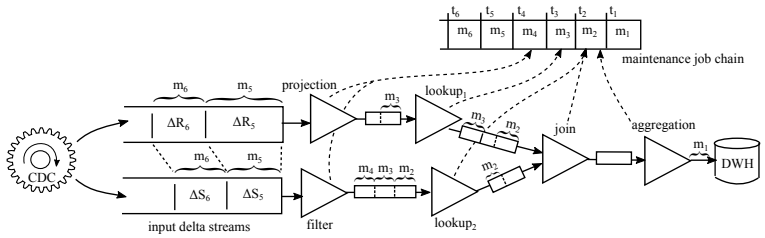


Fig. 1: Incremental ETL Pipeline

example of our incremental ETL pipeline and it consists of several components. The ETL pipeline is a data flow system and represented as a directed acyclic graph $G(V, E)$ where all nodes $v \in V$ (in triangle form) are ETL transformation operators and each edge $e \in E$ is an in-memory pipe used to transfer data from provider operator to consumer operator. The sources of the ETL pipeline are the input delta streams (one for each source table) that reside in the staging area. Each stream buffers source deltas (Δ: insertions (I), deletions (D) and updates (U)) which are captured by an independent change data capture (CDC) process and maintained in commit timestamp order. An event of a query arrival at timestamp t_i triggers the construction of a maintenance job m_{t_i} which groups the buffered source deltas with commit-time(Δ) < t_i and assigns them the id of this job. This job id is further sent to an auxiliary maintenance job chain as an id list of in-progress maintenance jobs. Each pipeline operator runs in a single, non-terminating thread and iterates through the id list. Given a job id, an operator thread blocks until input deltas with matching id occur. Once it finishes processing, it re-initializes itself and fetches the next pending job id from the job chain.

In this paper, we address pipeline scheduling for the consistency property while executing a chain of maintenance jobs using incremental ETL pipeline. We observed that data sets like staging tables or dimension tables can be read and written by different operations in the same pipeline. Take a logical incremental join (running multiple physical operators that need to access/refresh old state of the same join tables) as example. Without synchronizing or coordinating operator threads, anomalies can occur which breaks the consistency property in data warehouses. The remainder of this paper is structured as follows. We review existing scheduling algorithms from previous work mainly in ETL and stream processing domains in Section 2. In Section 3, we analyze the consistency anomalies in ETL pipelines, propose solutions called *consistency zones*, and discuss their implementations. Furthermore, we compare the experimental results of an existing scheduling algorithm (`MINIMUM COST`) with & without taking the consistency zones into account in Section 4.

2 Related Work

Previous research studies on workflow/pipeline scheduling in ETL & stream processing domains are mainly discussed here. Early work in stream processing domain addressed

scheduling algorithms for execution time, throughput or memory consumption purpose. Carney et al. [Ca03] presented their scheduling algorithms in their Aurora engine which assigns operators (from continuous queries) to CPU processors (i.e. threads) at runtime based on several metrics (e.g. selectivity, processing costs, thread context switches). To reduce the scheduling and operator overheads, they group operators into *superbox*es and traverse (i.e. thread assignment) operators in a superbox in a specific order that yields minimum context switches, a lower total execution time, or maximum memory utilization. Their scheduler decides which operator is assigned a thread to process how many tuples by estimating the processing cost (i.e. tuples per time unit). This is different from our case where all operators are initially started as Java threads and scheduled originally by operating system in a round robin, time-slicing manner. Application-level scheduling can be achieved by setting priorities to threads at runtime where threads with higher priorities get longer time quantum to process than those with lower priorities.

Karagiannis et al. [KVS13] also examined similar algorithms but in the (batch-oriented) ETL domain for throughput and memory consumption purposes. In their work, they exploit pipelining parallelism by dividing a large ETL workflow to connected subflows. Each subflow is a connected subgraph of original workflow and allows the data pipelining between operators. They further obtain a stratification of the subflow graph to assign mutually independent operators from subflows to subsequent layers of execution. Scheduling algorithms are applied in each subflow. For examples, their `MINIMUM COST` (MC) algorithm selects the operator with the largest volume of input data to first activate in a subflow. All work above suggested that continuous queries/ETL workflows are divided into subflows with operators connected with each other while in our work, operators in a so-called consistency zone can be separate and require to execute as an atomic group.

There exists also several work related to scheduling in data warehouses. Golab et al proposed scheduling algorithm based on the staleness metric of update jobs in streaming data warehouses [GJS12]. Resources are assigned to short/long jobs based on different policies. In [TPL08], authors introduced loading schemes to deliver trickle-updates in batch-load speed by buffering temporary incoming data either in memory on warehouse side or on client disks depending on system load. Thiele et al. [TFL09] introduced a model to schedule queries and updates at fine-grained data partition level for a balance between quality of service and quality of data in warehouses.

3 Operator Thread Coordination & Synchronization

As introduced in Section 1, the incremental ETL pipeline from our previous work is capable of handling multiple maintenance jobs simultaneously. However, potential consistency anomalies can occur in this model, which is addressed in this section for slowly changing dimensions and incremental join. Furthermore, we introduce two types of consistency zones as solutions to resolve these anomalies.

Slowly changing dimension (SCD) tables have different maintenance types. For example, SCDs of type 2 are history-keeping dimensions where multiple rows comprising the same

business key can represent a history of one entity while each row has a unique surrogate key in the warehouse and was valid in a certain time period (from start date to end date and the current row version has the end date null). With a change occurring in the source table of a SCD table, the most recent row version of the corresponding entity (end date is null) is updated by replacing the null value with the current date and a new row version is inserted with a new surrogate key and a time range (current date ~ null). However, the surrogate key of the old row version may concurrently be looked up in the fact table maintenance flow. Assume that the source tables, that are relevant to the maintenance of both fact tables and SCDs, reside in different databases. A globally serializable schedule *S* of the source actions on these source tables needs to be replayed in ETL flows for strong consistency in data warehouses [ZGMW96]. Otherwise, a consistency anomaly can occur which will be explained in the following (see Figure 2). At the upper-left part of Figure 2, two source

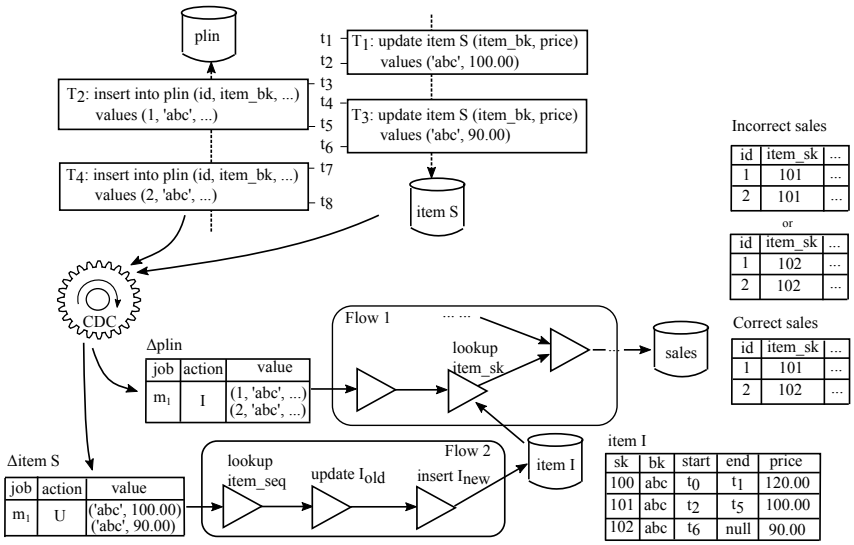


Fig. 2: Anomaly Example for ETL Pipeline Execution without Coordination

tables: `plin` and `item-S` are used as inputs for a fact table maintenance flow (`Flow 1`) and a dimension maintenance flow (`Flow 2`) to refresh warehouse tables `sales` and `item-I`, respectively. Two source-local transactions T_1 (start time: t_1 ~ commit time: t_2) and T_3 (t_4~t_6) have been executed on *item-S* to update the price attribute of an item with business key ('abc') in one source database. Two additional transactions T_2 (t_3~t_5) and T_4 (t_7~t_8) have been also completed in a different database where a new state of source table *plin* is affected by two insertions sharing the same business key ('abc'). Strong consistency of the warehouse state can be reached if the globally serializable schedule *S*: $T_1 \leftarrow T_2 \leftarrow T_3 \leftarrow T_4$ is also guaranteed in ETL pipeline execution. A consistent warehouse state has been shown at the bottom-right part of Figure 2. The surrogate key (101) found for the insertion (`1, 'abc', ...`) is affected by the source-local transaction T_1 on *item-S* while the subsequent insertion (`2, 'abc', ...`) will see a different surrogate key (102) due to T_3. However, the input delta streams only reflect the local schedules S_1: $T_1 \leftarrow T_3$ on *item-S* and S_2: $T_2 \leftarrow T_4$ on *plin*. Therefore, there is no guarantee that the global schedule *S* will be correctly replayed

since operator threads run independently without coordination. For example, at time t_9, a warehouse query occurs, which triggers an immediate execution of a maintenance job m_1 that brackets T_2 and T_4 together on *plin* and groups T_1 and T_3 together on *item-S*. Two incorrect states of the *sales* fact table have been depicted at the upper-right part of the figure. The case where *item_sk* has value 101 twice corresponds to an incorrect schedule: $T_1 \leftarrow T_2 \leftarrow T_4 \leftarrow T_3$ while another case where *item_sk* has value 102 twice corresponds to another incorrect schedule: $T_1 \leftarrow T_3 \leftarrow T_2 \leftarrow T_4$. This anomaly is caused by an uncontrolled execution sequence of three read-/write-operator threads: *item_sk-lookup* in Flow 1 and (*update-*I_{old}, *insert-*I_{new}) in Flow 2.

The potential anomaly in **incremental join** is explained here. An incremental join is a logical operator which takes the deltas (insertions, deletions and updates) on two join tables as inputs and calculates target deltas for previously derived join results. In [BJ10], a delta rule was defined for incremental joins. Let ΔR denote insertions on table R. As shown below, given the old state of the two join tables (R_{old} and S_{old}) and corresponding insertions (ΔR and ΔS), new insertions affecting previous join results can be calculated by first identifying matching rows in the mutual join tables for the two insertion sets and further combining the incoming insertions found in ($\Delta R \bowtie \Delta S$). For simplicity, we use the symbol Δ here to denote all insertions I, deletions D and updates U. Hence, the rule applies to all three cases with an additional join predicate (`R.action = S.action`) added to ($\Delta R \bowtie \Delta S$), where action $\in$ {I, D, U}.

$$\Delta(R \bowtie S) \equiv (\Delta R \bowtie S_{old}) \cup (R_{old} \bowtie \Delta S) \cup (\Delta R \bowtie \Delta S)$$

We see that a logical incremental join operator is mapped to multiple physical operators, i.e. three join operators plus two union operators. To implement this delta rule in our incremental ETL pipeline, two tables R_{old} and S_{old} are materialized in the staging area during historical load and two extra update operators (denoted as $\uplus$) are introduced. One $\uplus$ is used to gradually maintain the staging table S_{old} using the deltas ($\Delta_{m_1}S, \Delta_{m_2}S, \ldots \Delta_{m_{i-1}}S$) from the execution of preceding maintenance jobs ($m_1, m_2, \ldots, m_{i-1}$) to bring the join table S_{old} to the *consistent* state $S_{m_{i-1}}$ for $\Delta_{m_i}R$:

$$S_{m_{i-1}} = S_{old} \uplus \Delta_{m_1}S \ldots \uplus \Delta_{m_{i-1}}S = S_{old} \uplus \Delta_{m_{1\sim(i-1)}}S$$

Another update operator $\uplus$ performs the same maintenance on the staging table R_{old} for $\Delta_{m_i}S$. Therefore, the original delta rule is extended in the following based on the concept of our maintenance job chain.

$$\begin{aligned}\Delta_{m_i}(R \bowtie S) &\equiv (\Delta_{m_i}R \bowtie S_{m_{i-1}}) \cup (R_{m_{i-1}} \bowtie \Delta_{m_i}S) \cup (\Delta_{m_i}R \bowtie \Delta_{m_i}S) \\ &\equiv (\Delta_{m_i}R \bowtie (S_{old} \uplus \Delta_{m_{1\sim(i-1)}}S)) \cup ((R_{old} \uplus \Delta_{m_{1\sim(i-1)}}R) \bowtie \Delta_{m_i}S) \cup (\Delta_{m_i}R \bowtie \Delta_{m_i}S)\end{aligned}$$

The deltas $\Delta_{m_i}(R \bowtie S)$ of job m_i are considered as *consistent* only if the update operators have completed job $m_{(i-1)}$ on both staging tables before they are accessed by the join operators. However, without further precautions, our ETL pipeline only ensures that the maintenance job chain is executed in sequence in each operator thread. Inconsistencies can occur when directly deploying this extended delta rule in our ETL pipeline runtime. This is due to concurrent executions of join and update operators on the same staging table for different jobs.

We use a simple example (see Figure 3) to explain the potential anomaly. The two staging tables `Customer` and `Company` are depicted at the left-upper part of Figure 3 which both have been updated by deltas from m_1. Their input delta streams are shown at left-bottom part and each of them contains a list of tuples in the form of (`job, action, value`) which is used to store insertion-/deletion-/update-delta sets (only insertions with action I are considered here) for each maintenance job. Logically, by applying our extended delta

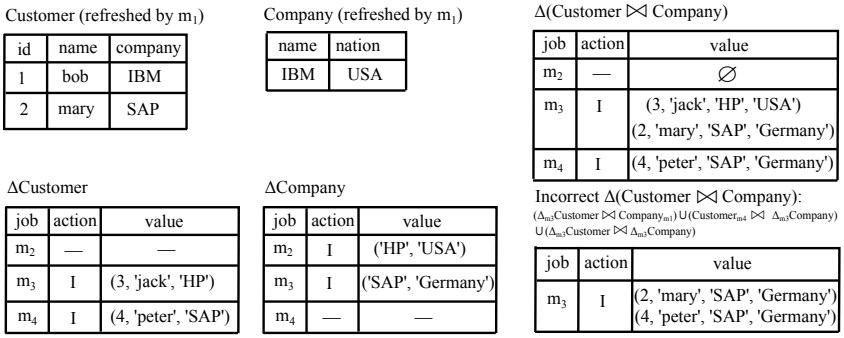


Fig. 3: Anomaly Example for Pipelined Incremental Join

rule, consistent deltas Δ(`Customer` ⋈ `Company`) would be derived which are shown at the right-upper part. For job m_3, a matching row (`'HP', 'USA'`) can be found from the company table for a new insertion (`3, 'jack', 'HP'`) on the customer table after the company table was updated by the preceding job m_2. With another successful row-matching between Δ_{m_3}Company and Customer$_{m_2}$, the final deltas are complete and correct. However, since each operator thread runs independently and has different execution latencies for inputs of different sizes, an inconsistent case can occur, which is shown at the right-bottom part. Due to differences in processing costs, the join operator Δ_{m_3}Customer⋈Company$_{m_1}$ has already started before the update operator completes m_2 on the company table and has mistakenly missed the matching row (`'HP', 'USA'`) from $m2$. And the other join operator Customer$_{m_4}$ ⋈ Δ_{m_3}Company accidentally reads a *phantom* row (`4, 'peter', 'SAP'`) from the maintenance job m_4 that is produced by the fast update operator on the customer table. This anomaly is caused by a pipeline execution without synchronization of read-&write-threads on the same staging table.

Consistency zone is a subgraph of the original flow graph. Operator nodes in a consistency zone do not have to be interconnected via data pipes while they always affect the same *shared mutable objects* (e.g. dimension/staging tables). To change the state of shared mutable objects using a chain of maintenance jobs in a consistent manner, the completeness of a maintenance job is synchronized in a zone while the actual execution sequence of inner nodes depends of the type of the consistency zone, e.g. in a specific order or in parallel. The concept of consistency zone is similar to nested transactions.

Pipelined Slowly Changing Dimension aims at a correct globally serializable schedule S. The CDC component participates in rebuilding S by first tracking start or commit timestamps of source-local transactions[3], mapping them to global timestamps and finally

[3] Execution timestamps of in-transaction statements have to be considered as well, which is omitted here.

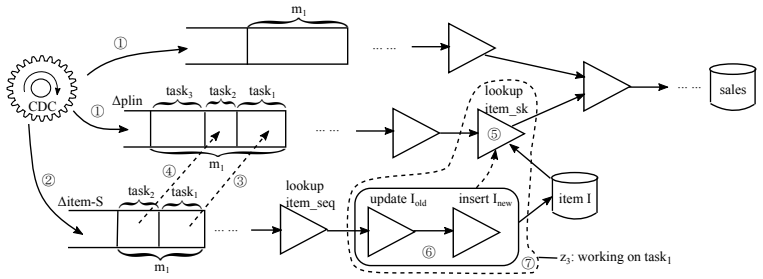


Fig. 4: Pipelined SCD with Consistency Zone

comparing them to find out a global order of actions. In addition, the execution of relevant operator threads needs to be coordinated in this global order in the incremental ETL pipeline. Therefore, a newly defined consistency zone is illustrated in Figure 4[4]. Recall that a maintenance job is constructed when a query is issued or when the size of any input delta stream exceeds a threshold (see Section 1). We refine the maintenance job into multiple internal, fine-grained *tasks* whose construction is triggered by a commit action of a source-local transaction affecting the source table of a SCD. As shown in Figure 4, ① the CDC continuously puts those captured source deltas into the input delta streams (one is Δ*plin*) of the fact table maintenance flow. At this time, a source-local update transaction commits on *item-S*, which creates a task_1 and comprises the delta tuples derived from this update transaction ②. This immediately creates another task_1 in the input delta stream Δ*plin* which contains all current available delta tuples ③. This means that all source-local, update transactions belonging to the task_1 in Δ*plin* have committed before the task_1 of Δ*item-S*. With a commit of the second update transaction on source table *item-S*, two new task_2 are created in both input delta streams ④. When a query is issued at a later time, a new m_1 is constructed which contains $\text{task}_{1\sim2}$ on Δ*item-S* and $\text{task}_{1\sim3}$ on Δ*plin* (delta tuples in task_3 commit after the task_2 in Δ*item-S*). During execution on m_1, a strict execution sequence between the atomic unit of *update-*I_{old} and *insert-*I_{new} and the *item_sk-lookup* is forced for each task_i in m_1. The *update-*I_{old} and *insert-*I_{new} have to wait until the *item_sk-lookup* finishes task_1 ⑤ and the *item_sk-lookup* cannot start to process task_2 until the atomic unit completes task_1 ⑥. This strict execution sequence can be implemented by the (Java) *wait/notify* methods as a provider-consumer relationship. Furthermore, in order to guarantee the atomic execution of both *update-*I_{old} and *insert-*I_{new} at task level, a (Java) *cyclic barrier*[5] object can be used here to let *update-*I_{old} wait to start a new task until *insert-*I_{new}

[4] It is worth to note that the current ETL tool does not provide direct implementation of the SCD (type 2) maintenance. To address this, we simply implement SCD (type 2) using *update-*I_{old} followed by *insert-*I_{new}. These two operator threads need to be executed in an atomic unit so that queries and surrogate key lookups will not see an inconsistent state or fail when checking a lookup condition. Another case that matters is that the execution of Flow 1 and Flow 2 mentioned previously is not performed strictly in sequence in a disjoint manner. Instead of using flow coordination for strong consistency, all operators from the two flows (for fact tables and dimension tables) are merged into a new big flow where the atomic unit of *update-*I_{old} *insert-*I_{new} operator threads can be scheduled with the *item_sk-lookup* operator thread at a fine-grained operator level.

[5] One cyclic barrier object (*cb*) can be embedded in those threads that need to be synchronized on the completion of the same job/task. Each time a new job/task starts , this *cb* object sets a local count to the number of all involved threads. When a thread completes, it decrements the local count by one and blocks until the count becomes zero.

completes the current one ⑥. Both thread synchronization and coordination are covered in this consistency zone ⑦.

Pipelined Incremental Join is supported by two *consistency zones* and an extra duplicate elimination operator. The consistency zone synchronizes the read-&write-threads on the same maintenance job and a new maintenance job is not started until all involving threads have completed the current one. Figure 5 shows the implementation of our pipelined incremental join. There are two consistency zones: z_1(update-R_{old}, $R_{old} \bowtie \Delta S$) and

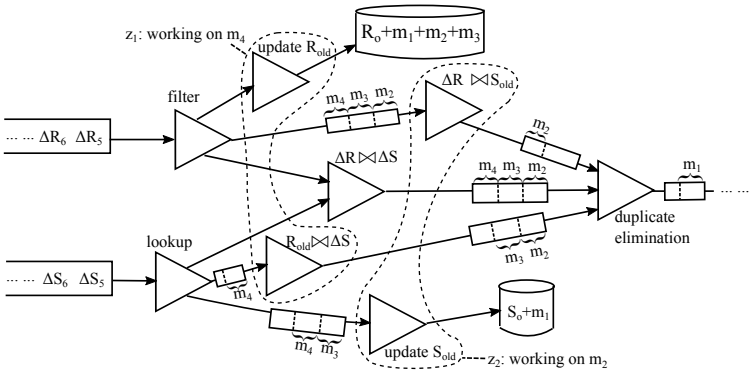


Fig. 5: Pipelined Incremental Join with Consistency Zones

z_2($\Delta R \bowtie S_{old}$, update-S_{old}), each of which maintains a (Java) *cyclic barrier* object to synchronize reads and writes on the same staging table for each job. The processing speeds of both threads in z_1 are very similar and fast, so both of them are currently working on m_4 and there is no new maintenance job buffered in any of the in-memory pipes of them. However, even though the original execution latency of the join operator thread $\Delta R \bowtie S_{old}$ is low, it has to be synchronized with the slow operator update-S_{old} on m_2 and a pile-up of maintenance jobs ($m_{2\sim4}$) exists in its input pipe. It is worth to note that a strict execution sequence of two read-/write threads is not required in a consistency zone (i.e. update-R_{old} does not have to start only after $R_{old} \bowtie \Delta S$ completes to meet the consistency requirement $R_{m_{i-1}} \bowtie \Delta_{m_i} S$). In case $R_{m_{i-1}} \bowtie \Delta_{m_i} S$ reads a subset of deltas from m_i (in R) due to concurrent execution of update-$R_{m_{i-1}}$ on m_i, duplicates will be deleted from the results of $\Delta_{m_i} R \bowtie \Delta_{m_i} S$ by the downstream duplicate elimination operator. Without a strict execution sequence in consistency zones, involved threads can be scheduled on different CPU cores for performance improvement. Furthermore, even though two consistency zones finish maintenance jobs in different paces, only the m_2 part of their outputs is visible to the downstream duplicate elimination operator since it currently works on m_2.

4 Consistency-Zone-Aware Pipeline Scheduling

According to the scheduling algorithm called `MINIMUM COST` (MC) [KVS13] described in Section 2, an ETL workflow is divided into subflows (each of which allows pipelining operators) and the operator having the largest volume of input data is selected to execute in each subflow. In Figure 5, a possible fragmentation results in following operator

groups[6]: (filter, update-R_{old}, $\Delta R \bowtie S_{old}$), (lookup, $R_{old} \bowtie \Delta S$, update-S_{old}), ($\Delta R \bowtie S_{old}$) and (duplication elimination). However, in incremental ETL pipeline, efficiency can degrade due to synchronized execution of threads in consistency zones. The performance of a very fast pipelined subflow can drop significantly if one of its operators hooks a separate slow operator in a consistency zone outside this subflow. The side effect of consistency zones determines that they perform like blocking operations. Hence, all operators in consistency zones should be grouped together to new subflows as (update-R_{old}, $R_{old} \bowtie \Delta S$), ($\Delta R \bowtie S_{old}$, update-S_{old}), etc., which is called consistency-zone-aware MC.

Experiments: we compared the original MC and consistency-zone-aware MC here and examine the latencies of maintenance jobs in two system settings where input delta streams have a low or high input ratio, respectively (system load reaches its limit with a high input ratio). TPC-DS benchmark (www.tpc.org/tpcds) was used for experiments. The testbed comprised the fact table *store sales* (of scale factor 1), surrounding dimension tables and two staging tables materialized during historical load for pipelined incremental join. The data set is stored in a Postgresql (version 9.5) on a remote machine (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8GB RAM). Two maintenance flows (used to maintain the *store sale* fact table and the *item* dimension table) were merged into an incremental ETL (job) pipeline (see Figure 5) that ran locally (Intel Core i7-4600U Processor, 2×2.10 GHz, 12GB RAM) in our pipeline engine which is extended from the original Pentaho Kettle (version 4.4.3, www.pentaho.com) engine. A local CDC thread[7] simulated a low input ratio (150 tuples/s) and a high input ratio (700 tuples/s), respectively. Besides, another thread continuously issued queries to the warehouse, which triggered the constructions of maintenance jobs in random time intervals. In each setting with different scheduling policies, we collected the execution time as job latency (in seconds) for 70 maintenance jobs.

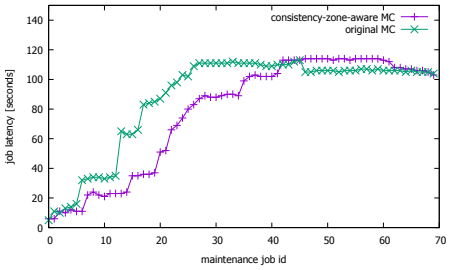


Fig. 6: input delta ratio: 150 tuples/s

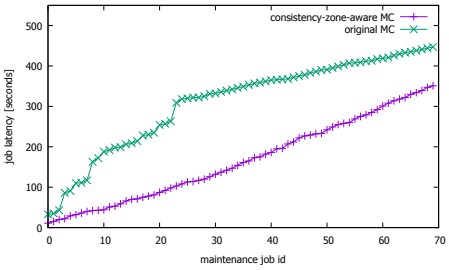


Fig. 7: input delta ratio: 700 tuples/s

Both figure 6 and 7 show that original MC performs worse than consistency-zone-aware MC, especially under high input ratio. The reason why original MC performs slow is due to the fact that the processing cost of the lookup operator was much slower, which causes starvations of downstream $R_{old} \bowtie \Delta S$ and update-S_{old}, as they are grouped in the same subflow as defined in original MC. However, the input pipes of update-R_{old} and $\Delta R \bowtie S_{old}$ grow drastically since they block due to our consistency zone features. More time quanta were assigned to them, which is not necessary and reduces the processing quantum of the

[6] blocking operations are subflows of their own.
[7] ran continuously to feed the input delta streams with source deltas to update the store sales and item table.

slow lookup operator. Hence, our consistency-zone-aware MC addresses this problem and groups the threads in consistency zones together to execute.

5 Conclusion

Based on an incremental ETL pipeline engine, we explained the potential consistency anomalies for incremental joins and slowly changing dimensions using easy-to-understand examples and proposed consistency zones with appropriate implementations. Furthermore, we moved a step towards extending previous scheduling algorithm with consistency zone features using experiments. We leave a detailed validation of further scheduling algorithms as future work.

References

[BJ10] Behrend, Andreas; Jörg, Thomas: Optimized incremental ETL jobs for maintaining data warehouses. In: Proceedings of the Fourteenth International Database Engineering & Applications Symposium. ACM, pp. 216–224, 2010.

[Ca03] Carney, Don; Çetintemel, Uğur; Rasin, Alex; Zdonik, Stan; Cherniack, Mitch; Stonebraker, Mike: Operator scheduling in a data stream manager. In: Proceedings of the 29th international conference on Very large data bases-Volume 29. VLDB Endowment, pp. 838–849, 2003.

[GJ11] Golab, Lukasz; Johnson, Theodore: Consistency in a Stream Warehouse. In: CIDR. volume 11, pp. 114–122, 2011.

[GJS12] Golab, Lukasz; Johnson, Theodore; Shkapenyuk, Vladislav: Scalable scheduling of updates in streaming data warehouses. IEEE Transactions on knowledge and data engineering, 24(6):1092–1105, 2012.

[KVS13] Karagiannis, Anastasios; Vassiliadis, Panos; Simitsis, Alkis: Scheduling strategies for efficient ETL execution. Information Systems, 38(6):927–945, 2013.

[Qu15] Qu, Weiping; Basavaraj, Vinanthi; Shankar, Sahana; Dessloch, Stefan: Real-Time Snapshot Maintenance with Incremental ETL Pipelines in Data Warehouses. In: International Conference on Big Data Analytics and Knowledge Discovery. Springer, pp. 217–228, 2015.

[TFL09] Thiele, Maik; Fischer, Ulrike; Lehner, Wolfgang: Partition-based workload scheduling in living data warehouse environments. Information Systems, 34(4):382–399, 2009.

[TPL08] Thomsen, Christian; Pedersen, Torben Bach; Lehner, Wolfgang: RiTE: Providing on-demand data for right-time data warehousing. In: 2008 IEEE 24th International Conference on Data Engineering. IEEE, pp. 456–465, 2008.

[ZGMW96] Zhuge, Yue; Garcia-Molina, Hector; Wiener, Janet L: The Strobe algorithms for multi-source warehouse consistency. In: Parallel and Distributed Information Systems, 1996., Fourth International Conference on. IEEE, pp. 146–157, 1996.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 309

A Modular Approach for Non-Distributed Crash Recovery for Streaming Systems

Michael Brand[1] Marco Grawunder[2] H.-Jürgen Appelrath†[3]

Abstract: High availability and reliability are important aspects for streaming systems. State-of-the-art recovery techniques like active or passive standby use several nodes to fulfill these requirements. But not every processing task is done in a professional environment with resources for a distributed system (e.g. smart home). Additionally, even distributed streaming systems can benefit from reliable non-distributed crash recovery (NDCR) because it can help to restore the overall system state faster not only after a node failure but also after the roll-out of updates.

In this paper, we present our research on NDCR for streaming systems and point out its possibilities and limitations. We propose a flexible and extensible framework in which small NDCR tasks can be combined to high-level NDCR classes with different degrees of fulfillment for completeness and correctness: at-most-once, at-least-once or exactly-once. Further, we offer a way to mark elements that may be incorrect or duplicated.

Keywords: Streaming System, Non-Distributed, Availability, Crash Recovery, Framework, Odysseus

1 Motivation

If a streaming system is deployed, it should be robust and act comprehensibly even if system failures occur. Unfortunately, system failures lead to the loss of all data within the volatile main memory and streaming systems typically hold all data in the main memory. Mechanisms to recover from system failures are called crash recovery. Despite the fact that streaming systems hold their data in the main memory, there is another major difference to other data processing systems like DBMS. Streaming systems process data from active data sources that will not stop sending if the streaming system crashes. That makes data loss unavoidable without a backup of the incoming data streams. State-of-the-art systems use distributed solutions with recovery mechanisms like active or passive standby to achieve high availability. For an industrial or commercial use of streaming systems, this is a desirable solution, but streaming systems can also be used in non-industrial and non-commercial environments, e.g. in combination with the Internet of Things. An example is the monitoring and control of smart devices in a smart home. Private households often have no computer cluster for their streaming/smart home tasks and uploading the data in the cloud for data analysis is often also not favored because of privacy concerns. In such a scenario, there is a need for recovery mechanisms that at least reduce the data loss due to a system failure without having several streaming system nodes. Additionally, a system failure is not the

[1] Universität Oldenburg, Abteilung Informationssysteme, michael.brand@uni-oldenburg.de
[2] Universität Oldenburg, Abteilung Informationssysteme, marco.grawunder@uni-oldenburg.de
[3] Universität Oldenburg, Abteilung Informationssysteme

only scenario for recovery mechanisms. Another scenario is the roll-out of updates as it typically includes the restart of the streaming system. After a restart, the streaming system should automatically continue the stream processing tasks as soon as possible and reach the same state as if there were no processing gap.

Distributed streaming systems can also benefit from recovery mechanisms for stand-alone streaming systems. A crashed streaming system node that is recovered with stand-alone recovery mechanisms will be available again in less time to take part in a failsafe distributed stream processing. The recovered node can for example act as a backup for the former backup node that is now the active one. As for stand-alone streaming systems, it is also possible to use stand-alone recovery after the roll-out of updates. In order to keep the stream processing tasks running, the nodes in a distributed streaming system can be updated at different times. If a backup node gets updated, it should synchronize with the active node automatically and as soon as possible so that the active node can be updated as well. Additionally, the risk of data loss rises with the time needed to synchronize if there is only one backup node because the active node could crash during that time.

Despite state-of-the-art recovery works in a distributed way, we focus on recovery mechanisms for stand-alone streaming systems that we call *Non-Distributed Crash Recovery (NDCR)*. Such an NDCR covers at least the following, but typically even more, tasks in order to minimize the data loss:

1. Detect a system failure.
2. Reinstall all connections to data sources and sinks.
3. Reinstall and restart all queries.

There are different requirements for an NDCR depending on the application scenario. Some applications may tolerate data loss or duplicates, other may need full correctness of the results. To see if and how an NDCR can meet these different requirements, we adopt a classification for recovery in distributed streaming systems from Hwang et al. [H+05]. They distinguish between three different classes of recovery for distributed streaming systems. The weakest recovery class is called *gap recovery* and it ensures neither input nor output preservation. Therefore, a system failure results in missing results for all incoming elements before the streaming system can process them again (*at-most-once delivery*). Gap recovery addresses the needs of applications that operate solely on the most recent information (e.g. sensor-based environment monitoring) where dropping old data is tolerable for reduced recovery time and run-time overhead. Because elements may be aggregated, missing elements may also result in incorrect output values [H+05]. That can be a major disadvantage depending on the size and type of windows (types may be time window, e.g. the last 10 seconds, or element window, e.g. the last 10 elements).

Rollback recovery is the second recovery class and ensures that failures do not cause information loss. More specifically, it guarantees that the effects of all input elements are always forwarded to the instances that continue processing the elements despite failures (*at-least-once delivery*). Achieving this guarantee requires input preservation. Therefore, rollback recovery techniques result in a complete output stream (relative to its failure-free

counterpart), but typically they also result in duplicate values (all values between the last checkpoint and the system failure) [H+05]. The strongest recovery class, which is called *precise recovery*, completely masks failures and ensures that the output produced by an execution with failure and recovery is identical to the output produced by a failure-free execution (*exactly-once delivery*). Achieving this guarantee requires input preservation as well as output preservation. Every rollback recovery technique (e.g. passive standby) can fulfill the requirements for precise recovery if all duplicates that are a result of rollback recovery are eliminated [H+05].

Obviously, the possibilities of an NDCR to fulfill the requirements are not the same as for distributed streaming systems: all incoming elements that are sent by the data sources while the streaming system is not available must be processed or at least hold available by another entity to make completeness and correctness achievable. In a distributed streaming system, this can be done using another node but for non-distributed streaming systems other concepts are needed.

In this paper, we present our novel research on *NDCR for streaming systems* with the following contributions:

1. We adopt the recovery classification proposed by Hwang et al. [H+05] to non-distributed streaming systems.
2. We point out the possibilities and the limitations of an NDCR.
3. We propose a flexible and extensible framework in which small recovery tasks can be combined to high-level recovery techniques. By doing so, we allow new combinations to fulfill individual recovery requirements.
4. We propose a way to mark elements that may be incorrect due to recovery (wrong values after a gap or duplicates) because in our opinion it is crucial to make the subsequent faults of a system failure transparent for users and other systems. For example, results of an aggregation may be different (compared to a failure-free run) because of missing elements or duplicates.
5. Our evaluation shows the different costs in terms of latency and throughput for the respective recovery technique.

We implemented the recovery tasks in *Odysseus* [A+12], an open-source streaming system, as compositions of plug-ins that make the recovery component flexible and extensible. As a result of this flexible implementation, we can compose other recovery techniques than those with a minimum overhead (but maximum gap), full completeness or full correctness (e.g. a recovery technique that saves and restores operator states but does not preserve the input streams).

The remainder of this paper is structured as follows. In Section 2, we give a brief overview of related work. The following sections present the respective recovery classes for non-distributed streaming systems: gap recovery in Section 3, rollback recovery in Section 4 and precise recovery in Section 5. The setup and results of our evaluation are provided in Section 6. Finally, Section 7 concludes the paper and gives a brief overview of future work.

2 Related Work

Hwang et al. [H+05] discuss different requirements for recovery of a distributed streaming system and they propose three different classes of recovery techniques (cf. Section 1). *Borealis* [A+05] uses a gap recovery technique, called Amnesia, that restarts a failed query from an empty state and continues processing elements as they arrive. Additionally, it uses several rollback recovery techniques with extensions for precise recovery (e.g. passive standby in which each processing node periodically sends the delta of its state to a backup node that takes over from the latest checkpoint if the processing node fails).

Another gap recovery technique is proposed by Dudoladov et al. [D+15] for iterative data flows. It eliminates the need to checkpoint the intermediate state of an iterative algorithm in certain cases. In the case of a failure, Dudoladov et al. use a user-supplied so-called compensation function to transit the algorithm to a consistent state from which the execution can continue and successfully converge. This optimistic recovery has been implemented on top of Apache Flink[4] for demonstration.

OSIRIS-SE [BSS05] is a stream-enabled hyper database infrastructure with two main characteristics: dynamic peer-to-peer process execution where reliable local execution is possible without centralized control as well as data stream management and process management in a single infrastructure. Brettlecker et al. [BSS05] define reliability in the context of data stream processing: stream processes have to be executed in a way that the process specification is met even in the case of a failure. Further, they introduce a classification of failures with a class called *service failures* among others. A service failure indicates that at least one operator is no longer available. That includes system failures of individual nodes. For temporary failures, OSIRIS-SE uses buffering techniques while it uses backup nodes for permanent failures.

S-Store [M+15] is an extension of H-Store[5], an open-source, in-memory, distributed OLTP database system. It attempts to fuse OLTP and streaming applications. S-Store implements streams as well as windows as time-varying H-Store tables and uses triggers for a push-based processing. Because of that approach, Meehan et al. [M+15] declare stream states and window states in S-Store to be persistent and recoverable with H-Store mechanisms. But they do not point out what happens with data sent while S-Store is crashed due to a system failure. However, S-Store provides two recovery mechanisms: Strong recovery guarantees to produce exactly the same state as that was present before the failure. Weak recovery will produce a consistent state that could have existed but that is not necessarily the same state as that was present before the failure. In our understanding of a streaming system and a data stream itself, the state changes between failure and recovery due to new elements from active data sources.

Storm [T+14] and *Heron* [K+15] use Zookeeper[6] to track the execution progress and to recover the state of a failed node. Both systems use, like the others mentioned before, the possibilities of a distributed system for recovery.

[4] flink.apache.org
[5] hstore.cs.brown.edu/
[6] zookeeper.apache.org

Apache Kafka [KK15] is a publish-subscribe system where producers publish messages to topics and consumers read messages in a topic. For horizontal scalability, a topic can be divided into partitions. Brokers assign an offset (a monotonically increasing sequence number per partition) to each message and store the messages on disk. A consumer polls messages of a topic in a partition sequentially from a broker and tracks the offset of the last seen message. The offset is periodically checkpointed to stable storage to support recovery of consumers. If Kafka is used with multiple brokers, each partition is replicated across the brokers to tolerate broker failures.

3 Gap Recovery

Gap recovery techniques do not preserve incoming data streams. Therefore, the data that is sent by the data sources after the system failure is lost until the streaming system can continue the stream processing (*at-most-once delivery*) [H+05]. Section 3.1 analyses the impacts of gap recovery on the stream processing results and Section 3.2 describes the concept of our modular framework that fulfills the requirements for gap recovery.

3.1 Impacts on the Stream Processing Results

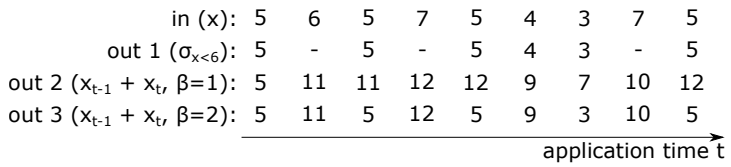
The length of a processing gap depends on four factors: the time needed to detect the failure, to correct the failure (especially for monolithic systems), to restart the streaming system (especially for monolithic systems), and the time needed for the recovery tasks (cf. Section 1). If the gap is measured in data stream elements (how many elements could the streaming system not process), the length of the gap depends also on the data rate of the incoming data streams [H+05]. For the scenario of an update (cf. Section 1), the first two factors are replaced by the time needed to update the streaming system.

Figure 1 illustrates the different possible impacts of a gap within incoming data streams. Figure 1a shows the incoming data stream *in* with simple numeric values. Three different output streams, *out* 1, *out* 2 and *out* 3, are the result of a simple filter operation and two aggregations respectively. *out* 1 contains all incoming elements that are less 6. *out* 2 sums each new input and the previous one up (sliding element window, slide $\beta = 1$). *out* 3 also sums the two elements up but without overlapping elements (tumbling element window, slide $\beta = 2$).

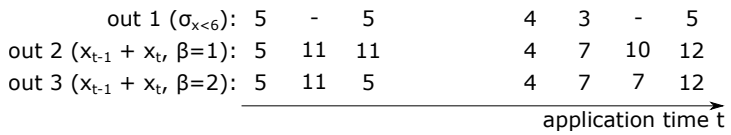
Figure 1b shows the same data stream processing but with a system failure causing a gap within the input stream. An obvious impact of the gap is a corresponding gap within the result streams, which we call *offline phase*.

Definition 1 (offline phase)**.**
The offline phase is the application time span between system failure and restart, in which elements are missing in the result stream and would be present in its failure-free counterpart.

For *out* 1, where each input element can be handled independently, the offline phase is the only impact. However, data stream operations often affect a set of data stream elements,



(a) Failure-free run.



(b) Run with system failure.

Fig. 1: A simple example for the different impacts of a gap within an incoming data stream.

e.g. relational aggregations like in *out* 2 and *out* 3 or joins. Typically, windows are used to define subsets of data stream elements that should be processed together, e.g. the average temperature of the last five minutes or the sum of the last two elements like in *out* 2 and *out* 3. Therefore, results that are not the same as their failure-free counterpart can occur after the offline phase. In Figure 1b, it is the first element after the offline phase in *out* 2. Here, the input value 4 and the previous one cannot be summed up because of the offline phase. Therefore, the result is different: 4 instead of 9. We call the period of results that may differ from their failure-free counterpart due to the offline phase *convergence phase*.

Definition 2 (convergence phase)**.**
The convergence phase is the application time span after an offline phase in which results exist but differ from results that would have been produced in the failure-free counterpart.

A gap within input streams can, therefore, cause a two-phase gap within result streams with an offline phase and a convergence phase. Whether a convergence phase exists depends on the operators and windows in the query plan as indicated by the example in Figure 1.

To distinguish different types of operators, we refer to the classification of Aurora operators that contains classes for arbitrary, deterministic, convergent-capable and repeatable operators [H+05]. Convergent-capable operators yield convergent recovery when they restart from an empty internal state and re-process the same input streams, starting from an arbitrary earlier point in time. Convergent recovery is a recovery that can result in a finite convergence phase after recovery (e.g. the sliding window in Figure 1). Hwang et al. [H+05] state that the window alignment is the only possible cause that prevents a deterministic operator from being convergent-capable (its window alignment does not converge to the same alignment when restarted from an arbitrary point in time). An example for an operator that is not convergent-capable is the aggregation with a tumbling window in Figure 1.

Convergent-capable operators are for their part *repeatable* if they yield repeating recovery when they restart from an empty internal state and re-process the same input streams, starting from an arbitrary earlier point in time [H+05]. Repeating recovery is a recovery that results in identical output elements after the offline phase compared to the failure-free

counterpart (no convergence phase). A necessary condition for an operator to be repeatable is to use at most one element from each input stream to produce a result (e.g. the filter in the example in Figure 1) [H+05]. The term "identical data streams" does not mean that every element must be at the same position in both data streams. The position can vary for elements with equal time stamps. Generally, we assume data streams to be ordered by time stamps.

Based on this operator classification, we conclude that a finite convergence phase exists if and only if the following constraints are fulfilled:

1. All operators are at least convergent-capable.
2. At least one operator is not repeatable.
3. The windows (failure run and failure-free run) do not contain the same elements.

Besides a finite convergence phase, there are two other options for a convergence: *immediate convergence* (no convergence phase) and *no convergence* (infinite convergence phase). Immediate convergence happens if all windows after the offline phase and their failure-free counterparts contain the same elements, or if all operators are repeatable and each element can be processed independently. For queries with operators that are not convergence-capable, the results may never converge. Non-determinism is one reason to have an infinite convergence phase (e.g. enrichment with randomized values). Another reason may be a shift of starting and ending points of windows because of the offline phase. An example is *out* 3 in Figure 1. It is the same sum operation like in *out* 2 but with tumbling windows with a slide of $\beta = 2$ instead of sliding windows with $\beta = 1$. If the first element after the offline phase is not a first summand in its failure-free counterpart, all sums after the offline phase will be calculated with other summands as their failure-free counterparts as in Figure 1b. Of course, it may also happen that there is no convergence phase after a system failure although not all operators are convergence-capable (e.g. non-deterministic behavior results in equal results by accident). Another aspect to mention is that not all data stream applications need a convergence of their results. A more detailed discussion about the types of convergences and the length of a convergence phase is outside the scope of this paper.

3.2 Modular implementation

The concept of our modular framework that fulfills the requirements for gap recovery provides several combinable subcomponents and is shown in Figure 2. Each subcomponent is responsible to back up particular information and to recover them if it is necessary. On the right, the persistent memory is illustrated. It is used by the subcomponents. Between the subcomponents and the persistent memory are different access layers integrated: *System Log*, *Processing Image* and *Backup of Data Streams (BaDaSt)*. A typical gap recovery technique consists of the five subcomponents at the top, *System State*, *Source and Sink Connections*, *Queries*, *Query States*, and *Window Alignments*, which will be explained below. The other subcomponents are needed for rollback and precise recovery. They will be explained in Section 4 and Section 5 respectively.

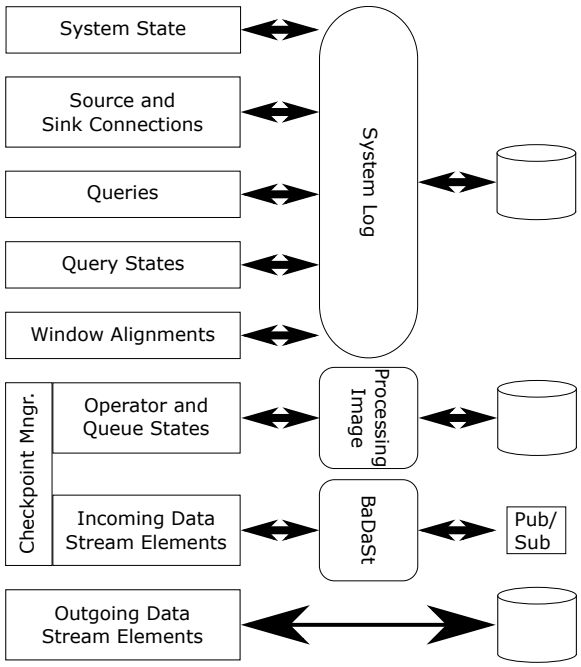


Fig. 2: Subcomponents of the NDCR to support various recovery techniques.

Before a streaming system can recover any information, it has to detect system failures. This can be done by logging all relevant system activities (e.g. the boot and shutdown processes). Therefore, we propose a *System Log* that is a protocol of rare system events that can be used to retrace relevant events. Additionally, it is possible to store other rarely changing events in the system log in order to recover them afterwards. The first subcomponent for gap recovery is the *System State* that has the following two functions:

1. Log successful boot and shutdown processes (write into system log).

2. While booting, search the system log for a missing shutdown event after the last boot event and notify the recovery component.

To fulfill the requirements of gap recovery (automatically continue processing after restart), it is necessary to take care of the connected data sources, the connected data sinks and the installed queries with their respective states. In this case, a query state means whether the query is running, paused, etc. All these aspects are typically rarely changing compared to data stream elements. This is why we propose to store this information in the system log. Therefore, the subcomponent *Queries* has the following three functions:

1. If a new query is installed, log the information that is necessary to reinstall it.

2. If a query is removed, log that information in order to prevent the installation of undesired queries.

3. While booting after a system failure (noticed by the subcomponent *System State*), search the system log for all queries that were installed after the last boot process and not removed afterwards and reinstall them.

The subcomponent *Source and Sink Connections* has similar functions. Closely related to the subcomponent *Queries* is the subcomponent *Query States* with the following functions:

1. If the state of a query change (e.g. paused, stopped, resumed), log that information in order to restore the latest query state.

2. For each reinstalled query (done by the subcomponent *Queries*), search the system log for the latest query state change and transform the query in the latest known state.

The best way to transform a query into its latest known state depends on the possibilities to switch from state to state and on the used streaming system. Intuitively, the functions of both subcomponents (*Queries* and *Query States*) could be done by a single subcomponent. We decided to separate them because reinstalling a query must be the first action for recovery of a query while recovering its state (e.g. start the query) must be done at the end. All other information to be recovered (e.g. operator states for other recovery techniques) should be done between reinstalling and restarting the query. Otherwise, the query processing would continue with empty operator states instead of recovered ones. Note in this context that gap recovery does not require the backup of operator states. Because of the offline phase, all operator states that were up-to-date before a system failure are typically outdated after system recovery (because of the progress in the incoming data streams).

The last subcomponent for gap recovery is called *Window Alignments*. As mentioned above, windows that do not contain the same data stream elements as their failure-free counterparts can cause a convergence phase. To limit the number of those "corrupted" windows (important for a finite convergence phase), there have to be windows that start after the offline phase and that contain the same data stream elements as their failure-free counterparts. To achieve that, all windows after recovery must have starting points that are also starting points in the failure-free run. For application-time windows with a constant slide (e.g. β *seconds*), it can be achieved by logging the starting point of the first window. All starting points after recovery can then be calculated using the logged starting point. Note that fixing the window alignment does typically not work for system-time windows and element windows. The content of a system-time window depends on execution-specific properties (e.g. how fast data is read). For element windows, the number of elements inside the offline phase is unknown if the data stream does not contain any counting information. Therefore, we only consider application-time windows for the subcomponent *Window Alignments*, but we intend to find solutions for element windows, too. The subcomponent *Window Alignments* has the following functions:

1. If a query with application-time starts processing, log the time stamp that is the starting point of the first window.

2. After reinstalling a query that uses application-time (done by the query state subcomponent), the starting points of the windows to come must be aligned with the logged starting point.

Consider the example in Figure 1b to clarify the functions of the subcomponent *Window Alignments*. The data source has a constant data rate of one element per second. *out* 3 uses application-time windows with a width of $\omega = 2\ seconds$ and a slide of $\beta = 2\ seconds$. The query starts and the window operator receives the first element with a time stamp ts_0. This means that all time stamps ts_i are starting points of windows for which $(ts_i - ts_0) \bmod 2\ seconds = 0$ is true. Without logging and recovering ts_0, the time stamp of the first element after the system restart would be interpreted as ts_0. But logging and recovering ts_0 results in correct starting points. Note in this context that the subcomponent *Window Alignments* is not necessary for every streaming system. If the time windows in a failure-free run are already aligned by using a constant point in time (e.g. $ts_0 = 0$), the alignment remains the same after a system failure.

Another important aspect for gap recovery is *syntactic transparency* that we define similar to Gulisano et al. [G+12] who define it for parallelization of queries.

Definition 3 (syntactic transparency)**.**
Syntactic transparency means that query recovery should be obvious to the user in a way that the user can identify results that may be not correct due to recovery.

To achieve syntactic transparency, we propose to annotate results that are within a convergence phase because they might not be equal to their failure-free counterparts. As we explained in this section, the convergence phase depends on the used operators and windows (we consider either some kind of time windows or element windows). The annotation that indicates whether an element is inside a convergence phase or not is defined as follows:

Definition 4 (trust)**.**
The annotation trust for an element is one of the three following values: `trustworthy`, which is the default value, `untrustworthy` for elements that are incorrect due to recovery and `indefinite` if it cannot be determined whether an element is incorrect due to recovery.

The trust annotation is a meta attribute of data stream elements that makes it necessary to merge several trust values if the corresponding elements are merged (e.g. in an aggregation or join). To maintain syntactic transparency, we implemented the merge function for the trust annotation as a min-function: if an element gets aggregated or joined with another element with a lower trust, its trust will be decreased to that lower trust. Using a merge function for the trust values allows for application-time windows to set only the trust of the first element after a system restart to `untrustworthy`. All results that are based on the first element after restart get the same reduced trust level. Results that are not based on the first element but on younger ones are after the convergence phase and their trust is not decreased. For element windows, it cannot be determined whether an element is inside a convergence phase or not. This is because the information about the number of elements that are in an offline phase (those we missed) is typically not available. Therefore, we propose to set the trust value to `indefinite` for all elements after gap recovery if element windows are used.

4 Rollback Recovery

Rollback recovery techniques can be used to avoid information loss because they guarantee completeness [H+05]. Section 4.1 analyses the impacts of rollback recovery on the stream processing results and Section 4.2 describes the components of our modular framework that are needed to fulfill the requirements for rollback recovery.

4.1 Impacts on the Stream Processing Results

We define *completeness* as follows:

Definition 5 (completeness)**.**
A complete result after recovery of a streaming operation contains all elements of its failure-free counterpart and no elements that its failure-free counterpart does not contain.

This requires input preservation, but the result streams after rollback recovery can also contain duplicates (*at-least-once delivery*) [H+05]. Hwang et al. [H+05] discuss rollback recovery for distributed streaming systems and we propose to assign their basic idea of rollback recovery to non-distributed streaming systems.

To achieve completeness, all elements that could not be processed due to a system failure have to be available after system recovery in order to process them. We call an application that is responsible for the backup of the incoming data streams *Backup of Data Streams (BaDaSt)*. Generally, there are two ways to put BaDaSt into practice offering some limitations for non-distributed streaming systems:

- Local: BaDaSt and the streaming system share resources (e.g. running on the same machine).
- Distributed: BaDaSt and the streaming system share no resources (they communicate via network).

The advantage of the distributed solution is the higher guarantee of completeness respectively the lower risk that a system failure impacts both streaming system and BaDaSt. Less communication costs and less hardware that has to be assigned to the data stream processing task are the advantages of the local solution. Besides these two ways of where to run BaDaSt, there are also different possibilities what application to use: specialized in-house development, publish/subscribe system, another streaming system, etc. We recommend to use the publish/subscribe system Apache Kafka [KK15] because it has already demonstrated that it can be a good solution to log data streams in a fault-tolerant and partitioned way [KK15]. For a local solution, a single Kafka broker can be used on the same machine, whereas a cluster of brokers on different machines can be set up for a distributed solution.

In order to reduce recovery time, it is not suitable to log all elements that have ever been received. As for database management systems [ÖV11], checkpoints should be used to limit the length of the logs.

As mentioned before, rollback recovery may result in duplicates. The elements that are duplicates are all elements that are logged by BaDaSt and have successfully been processed by the streaming system before a system failure occurred (all elements between the last checkpoint and the system failure). Whether the usage of BaDaSt is sufficient for rollback recovery, depends (as the convergence phase in Section 3) on the used operators and windows.

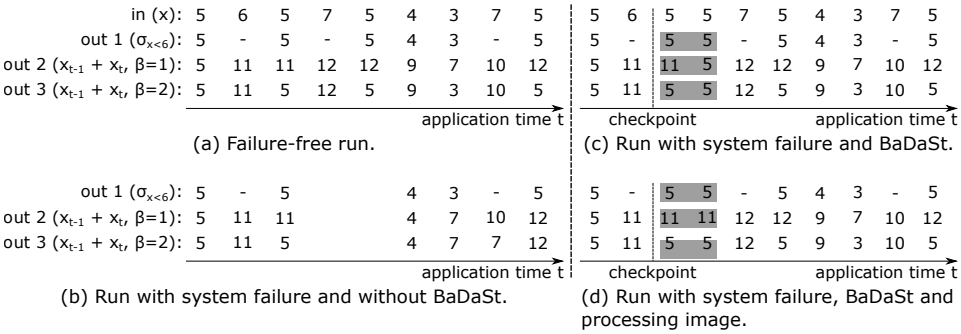


Fig. 3: The simple example (cf. Figure 1) enhanced for the impacts of BaDaSt and processing images.

Consider the same example as in Section 3 that is shown again in Figure 3a and Figure 3b. *in* is the incoming data stream, *out* 1 is the result of an operation that selects all elements lower than 6, and *out* 2 is the result of an operation that sums the current element and the last one up (sliding window). *out* 3 also sums the two elements up but without overlapping elements (tumbling window). Figure 3c illustrates the impacts of BaDaSt where all elements of *in* after a given checkpoint are logged. As one can see, the third element of *in* (5) appears twice because it is the only element that is between the checkpoint and the system failure (the gap). Since *out* 1 is the result of a repeatable operation, the third value of *out* 1 (5) is duplicated. *out* 2 is convergence-capable and it converges after one element because of the overlapping element windows (the current value is added to the last value). This results in the "wrong" value of 5, which would be 11 without system failure (cf. the third and fourth elements of *out* 2 that should be duplicates). *out* 3 is deterministic, but its third value (5) is correctly duplicated because it is the first element of the current tumbling window. To avoid convergence phases and therefore to reach completeness, a processing image is additionally needed for deterministic and convergence-capable operators.

Definition 6 (processing image)**.**
A processing image contains all operator and queue states in a query processing of a streaming system at a given point in time. An operator state consists of all information an operator stores for the next element to process (e.g. sums, counts). A queue state consists of all elements that are in the streaming system waiting to be processed by an operator (we assume streaming operators to be connected with queues as in [GÖ03]).

The points in time at which processing images are written should be the same as those at which the logs of BaDaSt start: the checkpoints. Figure 3d illustrates the impacts of a processing image together with BaDaSt. The processing image is written at the given checkpoint and all elements of *in* that have a time stamp after the checkpoint are logged. The

difference to Figure 3c is that the fourth element of *out* 2 is a duplicate because the partial sum (6) was stored as operator state. Therefore *out* 2 is complete (cf. *out* 2 in Figure 3a).

4.2 Modular implementation

To fulfill the requirements of rollback recovery, we propose several subcomponents to be built on top of the subcomponents for gap recovery. The new subcomponents are *Checkpoint Manager*, *Operator and Queue States*, and *Incoming Data Stream Elements* (cf. Figure 2). They use the already discussed *Processing Image* and *BaDaSt* as access layer. The *Checkpoint Manager* is a special subcomponent because it does not backup or recover information. It manages the checkpoints for the given stream processing task and notifies all interested entities (listeners) when a checkpoint is reached. The implementation of the *Checkpoint Manager* is easily exchangeable to allow different strategies (e.g. set a checkpoint every n elements, every n application time instants, or every n system time instants). To avoid concurrent modifications, the query processing has to be suspended if a checkpoint is reached. If the processing would continue while for example the processing image is stored, a correct image could not be guaranteed.

The processing image is stored and recovered by the subcomponent *Operator and Queue States* with the following functions:

1. If a checkpoint is reached, write for each stateful operator and queue its state in the processing image.
2. After reinstalling a query, load for each stateful operator and queue of that query its state from the processing image.

The second new subcomponent for rollback recovery handles backup and recovery of the *Incoming Data Stream Elements*:

1. If a new data source, whose data stream shall be logged, is connected, connect BaDaSt to that data source. From there on, BaDaSt stores all elements it receives.
2. If a checkpoint is reached, the streaming system determines the last received element of a data source and stores that information persistently (e.g. as an operator state).
3. After reinstalling a query, the streaming system connects to BaDaSt and sends the last element that has been received before the system failure. BaDaSt answers with all elements (in temporal order) that it has received after that element.

We propose to integrate a new *Source Recovery* operator in a query for the second and third function. This operator has the original data access operator and BaDaSt as inputs. It needs the element from the last checkpoint to recover after a system failure (cf. second function). First, it sends that element to BaDaSt in order to receive all stored elements that are in the temporal order after that one. From now on, the operator receives two input streams, one from the original data source and one from BaDaSt. To decide which element should be sent to the other streaming operators, the source recovery operator has three modes: `BaDaSt`, `Switch`, and `Source`. `BaDaSt` indicates that the elements from BaDaSt are older than the

elements from the data source (the current element from source is the watermark). If that is the case, the elements from BaDaSt will be transferred and the elements from the data source will be discarded. The operator switches to `Switch` if the elements from BaDaSt are not longer older as the elements from the data source. The last transferred element from BaDaSt becomes the watermark. `Switch` remains until we also receive the watermark from the data source. From this point in time on, we can transfer all subsequent elements from the data source without data loss (`Source`). Additionally, the *Source Recovery* operator can annotate all elements that are before the first element from the data source with a lower trust (`indefinite`, cf. Section 3) to indicate that they might be duplicates. Note that only those elements are duplicates that are received between the last checkpoint before the system failure and the system failure. All elements within the offline phase are no duplicates. But the option to annotate only the elements that are indeed duplicates is in this case not sufficient: if a duplicate is identified, it can be removed. That would be no more rollback but precise recovery.

As a conclusion of this section, it is also possible to deduce further recovery techniques with partial completeness guarantees (no 100% guaranteed completeness for all deterministic operations). An example of such a recovery technique has already been illustrated in Figure 3c. Here, the usage of BaDaSt without a processing image was sufficient for *out* 1 and *out* 3. This *stateless rollback recovery* is first of all useful for queries that consist solely of repeatable operations (*out* 1) because completeness can be guaranteed. Secondly, if tumbling (non-overlapping) time windows are used (*out* 3), the checkpoints can be set between consecutive windows. Then completeness can also be guaranteed. For other deterministic operations, the result is a convergence phase until all elements of a complete window are after the checkpoint. All later results that are calculated before the system failure are duplicates (*out* 2).

5 Precise Recovery

In contrast to gap or rollback recovery techniques, precise recovery techniques shall completely mask all failures and ensure semantic transparency (*completeness*, *exactly-once delivery*) [H+05]. In this context, we define *semantic transparency* closely linked to a definition by Gulisano et al. [G+12] for parallelization of queries:

Definition 7 (semantic transparency)**.**
Semantic transparency means that, for a given input, recovered queries should produce exactly the same output as their failure-free counterparts with the exception that, for elements with same time stamps, the order can differ.

Achieving this guarantee requires input preservation as well as output preservation. The only impact for the user is a temporary slower processing [H+05]. Hwang et al. [H+05] state that every rollback recovery technique can fulfill the requirements for precise recovery if all duplicates, which are the result of rollback recovery, are eliminated.

To eliminate duplicates, the recovery component needs to know those elements that the streaming system already sent to the data sinks (those that indeed left the system). One

possibility is to backup all outgoing data stream elements similar to the backup of the incoming data stream elements (cf. Section 4). Another more suitable solution is to track only the progress of a result stream. This is done by storing only the last sent element persistently (and override the older one every time). After system failure and rollback recovery techniques, the stored element can be used to identify all duplicates that are all elements before that element (incl. the element itself). For elements that are intended to be duplicates, additional information have to be logged in order to differentiate between them. This can be a simple counter for equal elements.

The new subcomponent *Outgoing Data Stream Elements* (cf. Figure 2) uses the persistent memory directly as data structure and has the following functions:

1. If a data stream element is sent by the streaming system as a result to a data sink, the information needed to identify that element is stored persistently in a file.
2. After reinstalling a query, the last stored information will be loaded from file. Before sending any new results, the results will be compared with that watermark and only newer elements will be sent.

With that extension to rollback recovery (cf. Section 4), the requirement for precise recovery (correctness) is fulfilled.

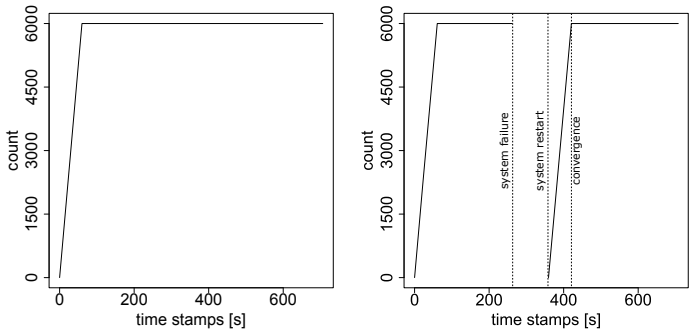
6 Evaluation

In this section, we present our evaluation results. Section 6.1 explains the evaluation setup, thus the used streaming system and query. In Section 6.2, we will show the completeness and correctness results for each recovery class and in Section 6.3 the syntactic transparency of our recovery architecture. Finally, we present the latency and throughput results in Section 6.4.

6.1 Setup

As mentioned before, we implemented our crash recovery architecture in *Odysseus* [A+12]. For our evaluation, we decided to choose a scenario that is easy to understand and to reproduce. The used data source sends elements with strictly monotonously rising time stamps every 10 ms. We installed a query in Odysseus [A+12] that counts the elements in sliding application-time windows with a width of $\omega = 60\ s$ and a slide of $\beta = 10\ ms$.

The expected result of a failure-free run is shown in Figure 4a. After starting the query, the count increases until the first window is complete (resulting in 6000 elements in a window for the used data rate). Afterwards, the count is constant due to the fact that for every new window, a new element is added and the oldest one is removed (slide of 10 ms matches the data rate). For the experiments with the recovery techniques, we let Odysseus [A+12] crash after 270 $seconds$ and we restarted it 90 $seconds$ later (both system-time). The checkpoints were set every 60 $seconds$ (system-time) and we used Apache Kafka [KK15] with a single, local broker for BaDaSt. Each experiment was repeated 10 times to get reasonable results.

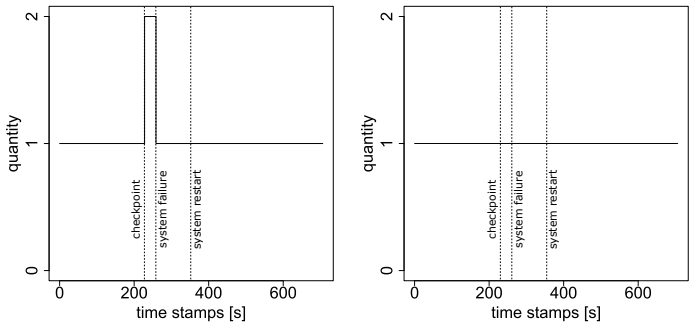


(a) Failure-free run. (b) Gap Recovery.

Fig. 4: The results of the counting aggregation.

6.2 Completeness and Correctness

In this subsection, we will analyze the completeness and correctness of the results of the counting aggregation. Figure 4b presents the results of the experiments with gap recovery. Before the system failure, all counts are the same as in the failure-free run. The offline phase lasts until the system restarts and is characterized by missing counts. Since no operator states (e.g. counts) are recovered, the count starts again with 0 after the offline phase and the results converge when the count is 6000 again. We omit to show the result stream for the experiments with rollback recovery respectively precise recovery. They are equivalent to the results of the failure-free counterpart because of their completeness (cf. Figure 4a).



(a) Rollback Recovery. (b) Precise Recovery.

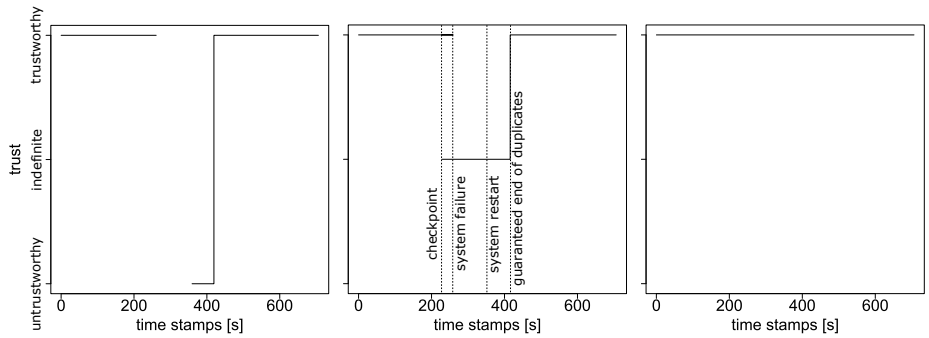
Fig. 5: The quantity of timestamps in the result stream.

To visualize the duplicate phase for rollback recovery, Figure 5a shows the quantity of each time stamp in the result stream for rollback recovery. In a failure-free run, each time stamp would occur exactly once, and in Figure 5a, all time stamps occur at least once (completeness), but there is a time span in which time stamps occur twice in the result streams: the duplicate phase between the last checkpoint and the system failure. Figure 5b

shows the quantity of the time stamps for precise recovery. Since all duplicates are eliminated, each time stamp occurs once (correctness).

6.3 Syntactic Transparency

As described in Section 3, we achieve syntactic transparency for our recovery architecture by annotating result streams with a trust value (`trustworthy`, `untrustworthy` or `indefinite`). Figure 6 shows the trust results for our experiments.



(a) Gap Recovery. (b) Rollback Recovery. (c) Precise Recovery.

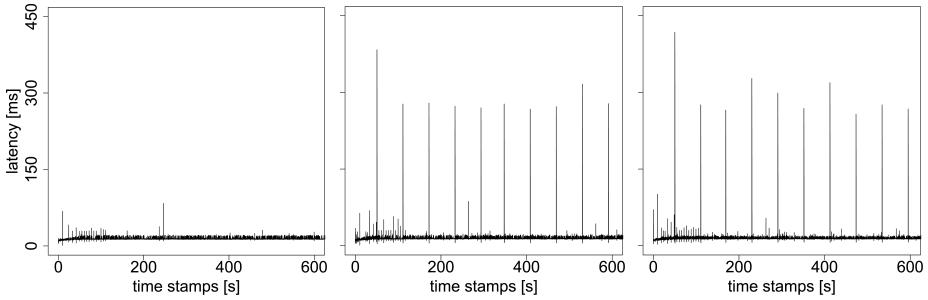
Fig. 6: The syntactic transparency for the results of our evaluation.

Figure 6a visualizes the trust for the gap recovery experiment. As in Figure 4b, the offline phase is evident. For the subsequent convergence phase, the trust is decreased to `untrustworthy` because the calculated counts are incorrect. Afterwards, the results are trustworthy again. Figure 6b shows the trust for the rollback recovery experiments. There is no gap within the trust values, but there are time stamps with two trust values. Those are the duplicates, each `trustworthy` before the system failure and `indefinite` after recovery (cf. Figure 5a). The trust is `indefinite` until the recovery component can guarantee that there are no more duplicates. This is when the streaming system receives the first element from the data source after system restart also from BaDaSt. The results in case of precise recovery shown in Figure 6c are always `trustworthy` (correctness).

6.4 Latency and Throughput

In the experiments for the latencies and throughputs, we used the backup mechanisms of the respective recovery class but without creating a system failure. Therefore, the experiments show the costs of the respective recovery class for its backup mechanisms.

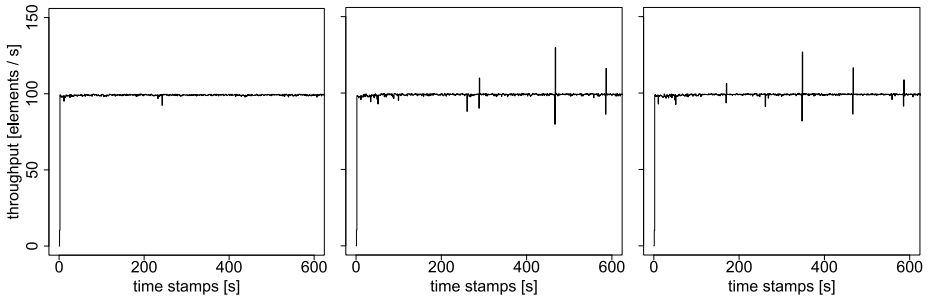
Figure 7 presents the latencies as the time between receiving the input element that triggered the output and sending the result. The latencies for the gap recovery experiments in Figure 7a and those of experiments without any backup mechanism are indistinguishable (we omit to show the latter). This is because there is no backup of information at the run-time of a query



(a) Gap Recovery. (b) Rollback Recovery. (c) Precise Recovery.

Fig. 7: The latency results of our evaluation.

if gap recovery is used (cf. Section 3). The median is approx. 13.33 *ms* and the average approx. 13.62 *ms*. For rollback recovery in Figure 7b, the latencies have peaks resulting in a higher average of 15.07 *ms*, but the median is almost the same (approx. 13.47 *ms*). The peaks match the checkpoint interval of 60 *s* (system-time) and the latencies increase because of the actions that are done: suspend processing, store processing image, and resume processing (cf. Section 4). The storing of each outgoing element is the difference between rollback recovery and precise recovery in Figure 7c, but with little effect on the latencies. The median is approx. 13.58 *ms* and the average approx. 14.97 *ms*.



(a) Gap Recovery. (b) Rollback Recovery. (c) Precise Recovery.

Fig. 8: The throughput results of our evaluation.

Figure 8 shows the throughputs as the number of incoming elements that can be processed per second. The impact of the backup mechanisms is low and the medians are approx. the following: 99.15 $elements/s$ for gap recovery in Figure 8a, 99.07 $elements/s$ for rollback recovery in Figure 8b and 99.06 $elements/s$ for precise recovery in Figure 8c. The theoretical maximum is 100 $elements/s$ (cf. data source in Subsection 6.1). The peaks (positive and negative) for rollback and precise recovery are caused by the suspension and resuming of the query at each checkpoint.

We evaluated the same query with a data source sending an element every millisecond. Due to place limitations we omit to show the results, but the costs for rollback and precise recovery increase dramatically (median up to 50 *s*). This is because Odysseus has to buffer much more elements if a checkpoint is reached. For such a scenario, a gap or a distributed recovery should be used.

7 Conclusion & Future Work

In this paper, we presented our flexible and extensible framework for an *NDCR for streaming systems*. We adopted a classification of recovery techniques for distributed streaming systems that contains three classes: gap recovery (at-most-once delivery), rollback recovery (at-least-once delivery) and precise recovery (exactly-once delivery). Since the results of gap recovery contain an offline phase and (dependent on operators and windows) a convergence phase, we calculate the length of a convergence phase based on application-time windows and annotate elements that are inside the convergence phase. That makes potentially incorrect results transparent to the user.

The limitations of NDCR are obvious because incoming elements have to be stored to achieve completeness (rollback recovery) and it has to be done outside the streaming system. One has to compromise about reliability and costs because the possibility increases that both streaming system and backup of data streams crash due to the same system failure with the number of shared resources. Additionally, processing images are needed in many cases for completeness. Similar to the elements inside a convergence phase, we annotate duplicates that are a result of rollback recovery with a decreased trust value. With an elimination of duplicates on top of rollback recovery, we achieve correctness (precise recovery).

We implemented the described recovery framework in the open-source streaming system *Odysseus* [A+12] and our evaluation results show the following conclusions:

1. The combinations of NDCR subcomponents to recovery techniques fulfill the respective requirements (completeness, correctness, syntactic transparency).
2. There are no run-time costs for gap recovery.
3. For rollback recovery, the costs in terms of increased latencies are only given for elements that are inside the streaming system while a checkpoint is reached.
4. It is not worth to use rollback recovery because the additional costs for precise recovery are low.
5. Rollback or precise recovery is only feasible for incoming data streams with moderate data rates or checkpoints in very small intervals.

The third point indicates that one has to compromise about backup overhead and recovery time when at least rollback recovery is used.

For future work, we intend to analyze the correlation between the used operators and windows on the one hand and the type of convergence phase (immediate convergence, finite convergence, no convergence) and its length on the other hand in more detail. For

element windows, we also intend to find a solution to avoid infinite convergence phases. Those can occur if the starting points of the element windows are not the same as for their failure-free counterparts. Finally, we plan to evaluate other recovery techniques such as *stateless rollback recovery* that should decrease the backup overhead.

References

[A+05] Ahmad, Y.; Berg, B.; Cetintemel, U., et al.: Distributed Operation in the Borealis Stream Processing Engine. In: Proceedings of the 2005 ACM SIGMOD. ACM, pp. 882–884, 2005.

[A+12] Appelrath, H.-J.; Geesen, D.; Grawunder, M., et al.: Odysseus – A Highly Customizable Framework for Creating Efficient Event Stream Management Systems. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems. ACM, pp. 367–368, 2012.

[BSS05] Brettlecker, G.; Schuldt, H.; Schek, H.-J.: Towards Reliable Data Stream Processing with OSIRIS-SE. In: Datenbanksysteme in Business, Technologie und Web (BTW) 2005. Pp. 405–414, 2005.

[D+15] Dudoladov, S.; Xu, C.; Schelter, S., et al.: Optimistic Recovery for Iterative Dataflows in Action. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 1439–1443, 2015.

[G+12] Gulisano, V.; Jimenez-Peris, R.; Patino-Martinez, M., et al.: StreamCloud: An Elastic and Scalable Data Streaming System. Parallel and Distributed Systems, IEEE Transactions on 23/12, pp. 2351–2365, 2012.

[GÖ03] Golab, L.; Özsu, M. T.: Issues in Data Stream Management. ACM Sigmod Record 32/2, pp. 5–14, 2003.

[H+05] Hwang, J.-H.; Balazinska, M.; Rasin, A., et al.: High-Availability Algorithms for Distributed Stream Processing. In: Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on. IEEE, pp. 779–790, 2005.

[K+15] Kulkarni, S.; Bhagat, N.; Fu, M., et al.: Twitter Heron: Stream Processing at Scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 239–250, 2015.

[KK15] Kleppmann, M.; Kreps, J.: Kafka, Samza and the Unix Philosophy of Distributed Data. IEEE Data Engineering Bulletin/, pp. 4–14, 2015.

[M+15] Meehan, J.; Tatbul, N.; Zdonik, S., et al.: S-Store: Streaming Meets Transaction Processing. In. 2015.

[ÖV11] Özsu, M. T.; Valduriez, P.: Principles of Distributed Database Systems. Springer Science & Business Media, 2011.

[T+14] Toshniwal, A.; Taneja, S.; Shukla, A., et al.: Storm @Twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, pp. 147–156, 2014.

Cloud and Benchmarks

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 331

Generating Data from Highly Flexible and Individual Process Settings through a Game-based Experimentation Service

Georg Kaes[1], Stefanie Rinderle-Ma[2]

Abstract: The ability to adapt process instances to changing requirements has long been recognized as a fundamental research topic. While in some settings, process flexibility is only required in exceptional situations, in other settings it is the key component which drives the process design. Examples can be found in multiple domains, including the nursing domain, where each patient requires his own, individual therapy process which may change on a regular basis. In this paper, such flexible and individual process settings (FIPS) are analyzed and the basic building blocks are defined based on expert interviews and relevant literature. The building blocks are then mapped onto a game-based experimentation service which offers a simulation and evaluation environment for FIPS. The data generated in this game are evaluated by a comparison with data from a real world FIPS.

Keywords: process flexibility, process logs, process mining, experimentation service

1 Introduction

Process flexibility is fundamental to many domains and thus a well researched topic [RW12]. Depending on the process setting there are different degrees of flexibility [Sc08]. In some settings, multiple process instances follow the same basic schema which has to be adapted if some exceptional situation occurs. In other cases, each process instance is different by design. Take the therapy process of a patient in a nursing home as an example (cf. [Ka14]). For each patient there exists one long-running process instance executing the therapy steps specific to this patient. There is no common schema, but each instance develops based on ad hoc adaptations. Whenever a patient shows new symptoms, the nurse has to adapt the therapy process accordingly. Such settings are referred to by *highly Flexible and Individual Process Settings (FIPS)*.

FIPS can be found in many domains [Ka14]. Examples range from regular customer care in a hotel setting, over the medical domain, to schools for children with special needs. In FIPS, end users are regularly supposed to decide on, specify, and conduct adaptations [We09]. Hence, system support becomes crucial [KR15]. In the nursing domain, for example, nurses could be supported in finding the best adaptation for the current patient's therapy process based on his current symptoms and medical history [Ka14]. Existing approaches on user support for process flexibility [Aa09; Gü08; KR15; We09] exploit information about previous process executions and adaptations, typically captured in process *execution* and

[1] University of Vienna, Faculty of Computer Science, Waehringerstrasse 29, 1090 Vienna, Austria, georg.kaes@univie.ac.at
[2] University of Vienna, Faculty of Computer Science, Waehringerstrasse 29, 1090 Vienna, Austria, stefanie.rinderle-ma@univie.ac.at

change logs. The goal is to learn from previously applied changes and to utilize additional knowledge such as context information, e.g., patient age and medical history, to identify the best adaption for the situation at hand.

Providing system-based user support requires to understand building blocks that are characteristic to FIPS independent of the particular domain. In [Ka14] requirements for highly flexible process settings have been collected from four different application domains, i.e., *subject data*, *environmental data*, *goals*, *trigger conditions*, and *process fragments*. The first question is whether these requirements can be generalized as building blocks for FIPS. A follow-up question is whether the building blocks can be mapped and integrated into an experimentation service. The goal of such a service would be to enable the simulation of FIPS independent of any application domain. Such an experimentation service would enable the simulation of FIPS with a holistic view on data. It would also enable the creation of process execution and change logs in case no real-world logs are available due to, for example, legal or privacy reasons.

Overall, this paper tackles the following research questions:

Q1 Which building blocks are common to FIPS (following up on [Ka14])?

Q2 How to reflect FIPS requirements in a game-based experimentation service?

Q3 Is a game-based design suitable for generating the data common to a FIPS?

The research questions are tackled by elaborating a service-based design of a tower defense game that simulates FIPS. In a tower defense game, players defend towers from incoming enemies. To do this, players can select from multiple defense systems which can be placed at the towers. These defenses then fight the incoming enemies. At an abstract level, players conduct comparable actions to nurses in a nursing home: They plan individual process adaptions in order to deal with a certain problem. In the nursing home, these problems are the symptoms a patient shows; in the tower defense game, they are the incoming enemies. While playing the game, process change and execution logs and related data for analyzing the logs are logged in a data repository. Additionally, the tower defense game itself provides a setting which can be used to evaluate approaches which support users in the sequel.

The paper is structured as follows: In Section 2, building blocks and concepts of a FIPS are analyzed based on expert interviews and explained including data sources and components which have to be represented in the tower defense game ($\rightarrow$ Q1). Section 3 describes the design of the service-based tower defense game: Here, we present the mapping of FIPS requirements, data sources, and components to the concepts of the tower defense game ($\rightarrow$ Q2). Further on, the service-based architecture of the game and details of the generated log files are presented. The feasibility of the data generated by the game is evaluated by comparing the generated data elements with data generated by a real world FIPS ($\rightarrow$ Q2, Q3). Finally, Section 5 discusses related work and Section 6 concludes the paper.

2 Generalization of Highly Flexible and Individual Process Settings

FIPS can be found in many domains. As shown in previous work [Ka14], the basic properties of such settings are quite similar. Following up on [Ka14] and considering further work on flexible processes [RW12; We09], we conducted several expert interviews with practitioners from different domains which we identified as possible FIPS. Our goal was to enhance the basic building blocks we found in [Ka14] and to identify basic properties of FIPS. In this section, we first present the resulting building blocks (Section 2.1), followed by the outcome of the expert interviews and individual properties of FIPS (Section 2.2).

2.1 FIPS Building Blocks

Based on related work [RW12; We09] and the results from the expert interviews, the following requirements are considered to build the basis for a general FIPS representation (→ Q1). These building blocks are domain independent abstractions from general concepts such as data sources or procedures which exist in specific domains which can be seen as FIPS. We found implementations of these concepts in any domain we investigated which we considered as a FIPS. The rest of the section defines these building blocks and gives examples from the nursing domain. Following, the domains will be described in detail and domain specific implementations of these concepts will be presented.

- **Subjects** define the individual persons, objects or concepts that are subject to the process execution [Ka14]. In the nursing domain, these individuals are the patients for which the process instance exists and is being adapted.
- **Process Instances** [RW12] capture and execute the process logic for a subject. In FIPS, instances typically run for a long time and are affected by process change operations whenever necessary. In the nursing domain, the process instances contain the therapy tasks which have to be executed in order to make the patient feel good or better.
- **Organizational Units** reflect the organizational perspective of a process. They can be used to define relationships between actors, define skills and possible actions for those units etc. [RW12]. In the nursing domain, the organizational units are the nurses, doctors, assistant nurses, and all other employees of the nursing home which are in some way related to a patient's therapy plan.
- The **Environment** [Ka14] of a FIPS comprises all data that is related to the subject, but not directly incorporated by the subject, for example, the limited resources which are required to execute the tasks related to the subject. In the nursing domain, the environment would be the nursing home.
- **Process Fragments** are parts of a process which get inserted into, deleted from, or moved during a process change operation [Ka14; RW12]. In the nursing domain, these process fragments refer to therapies which are added to or removed from a patient's therapy plan.

- **Problem List:** For each fragment a set of conditions where the fragment should not be used can be defined. In [We09], such a problem list item has been introduced in an example from a medical domain, where a certain adaptation could not be made because the patient has a cardiatric pacemaker.
- **Bonus List:** In contrast to the problem list, the bonus list for each fragment describes situations where the fragment is known to perform well. This building block has been identified during the expert interviews. In the nursing domain, this bonus list contains details about the patient which give a hint that a certain therapy may perform well.
- **Triggers:** In [RW12], the necessity of ad hoc modifications to a process instance is defined by exceptions which have not been thought of when designing the process model. Whenever such an exception occurs, the process instance has to be changed in order to deal with this exception. In FIPS, such events occur on a regular basis, thus we argue that the term *exception* does not fit in this context. Since these situations trigger a change operation, we will use the term *trigger* instead. In the nursing domain, symptoms are the triggers. Whenever a patient shows new symptoms, something has to be changed in his therapy process.
- **Positive Goals [KK97]:** There are always reasons which justify a process change. These reasons can be defined as *goals* which should be reached by executing the tasks which have been inserted, or not executing the tasks which have been removed from the process instance. In the nursing domain, positive goals can be to make the patient feel better, more confident, or at least to stabilize him in his current condition.
- **Negative Goals [KK97]:** Process changes can also be conducted in order to avoid certain situations and hence they address negative goals. In the nursing domain, examples include the deterioration of the patient's condition or even the death of the patient.

2.2 Expert Interviews

We conducted interviews with experts from different domains in order to identify the basic building blocks of a FIPS presented in the last section. Highly individual settings can be considered as FIPS if they require some kind of process to be adapted in order to react properly to certain situations. In [Ka14] the basic high-level components for FIPS were analyzed for four domains, i.e. *manufacturing*, *software development*, *hotel and event management* and *care planning*. For evaluating the building blocks and general properties of a FIPS, we chose *software development* and *hotel and event management* as domains already evaluated in [Ka14]. In order to emphasize the diversity of settings where FIPS plays a key role, we decided to evaluate two additional domains, namely a *special needs school* and the *PhD program* of a university.

The interviews started with the experts describing the domain they are working in. We asked them for examples where they had to deal with special individual situations, and what they could personally learn from these situations. In the following, we presented them the basic idea of FIPS based on the nursing scenario and discussed possible commonalities between

their domain and the nursing domain as a FIPS. We chose the nursing domain as a reference, since it is easy to empathize with, even if the interviewed person is no expert in this domain. Based on this information, we identified in cooperation with the domain experts the basic building blocks which have been presented in the last section. The exact allocation of each building block for each evaluated domain can be found on our project website[3].

2.2.1 Expert Interview: Software Sales and Support

We interviewed two sales managers from mesonic, an austrian company which develops enterprise resource planning (ERP) and customer relationship management (CRM) software for small and medium enterprises. The contact to the customers works to a big part over a network of retailers which are in geographical vicinity of their customers. The support of mesonic mainly works over an internal system where problems with the software itself (e.g., bug reports), individual wishes, and additional requirements from certain customers and retailers are gathered. Depending on the information the company's support department has about the case at hand, different measures are taken: If a bug is identified (which is already known or reported by many other customers), it will be discussed in the next developer meeting, where further steps will be defined. Depending on its priority, it may be solved right away, or as soon as resources are free. A customer's whish for an additional feature may be implemented in a future version of the product, depending on the workload of the developers, the usefulness to other customers, and several other parameters.

Discussion: After introducing the nursing domain as a point of reference, we talked about commonalities between the software sales and support and the nursing domain on an abstract level. It became clear that commonalities between the individual problems of a patient in the nursing home and the problem description in a support case exist indeed: In both scenarios, there is a problem which has to be solved. In order to identify the best course of action (i.e. the best process fragment), parameters from the case itself as well as from the environment are analyzed. While in the nursing home the circumstances of the patient's problem are analyzed, including his medical history and common diseases, in the software domain the circumstances of the support case are identified, including the targeted version of the software and other, similar reported problems. When the problem itself is identified and the goals are clear, the next steps are planned. In the software domain these steps include work to be done by software developers, the support and sales departments. After these steps have been conducted the person in charge of the case evaluates whether the goals have been reached or not.

2.2.2 Expert Interview: Hotel Management and Guest Care

Hotel management also deals with individual subjects (i.e., guests) having specific requirements. We interviewed two receptionists from a local hotel and seminar location which

[3] http://cs.univie.ac.at/project/apes

focuses on business as well as leisure guests. The guest profiles range from typical seminar groups with a trainer and multiple employees or managers, to couples or families who want to spend some days in a hotel. Obviously, these groups have different requirements for the location: While for the leisure guests, for example, outdoor activities or wellness offers have a higher priority, for the business guest a well-equipped seminar location or a smooth procedure of their workshop is more important. Hence, the hotel has to provide different offers depending on the guests' profiles and demands.

Discussion: Each interaction with a guest can be seen as part of a FIPS. Depending on the guest profile and the current offers of the hotel and seminar location, different things can be offered to the guests. For example, if a guest is known to be quite exhausting, he will be treated with special care in order to keep him at bay. A seminar group who is known to have problems with technical equipment will receive special support right from the start. This behavior again shows commonalities to the nursing or the software development domain: First, the situation and the problem are identified, then, based on knowledge about the subject at hand and what has worked in a similar situation before, the best course of action is identified.

2.2.3 Expert Interview: Special Needs School

We interviewed two teachers from a special needs school who teach children with special needs from ages 10 to 15. The special needs range from language issues, over physical up to psychical conditions of various kinds. Depending on the specific needs of the children different teaching concepts have to be chosen. While there exists a general plan for the school year as a whole, it has to be presented to each child in a different way. Of course, due to resource limitations, this is not always possible. Hence, the teachers are required to balance the needs of all children.

Whenever a child shows some ad hoc needs, for example his or her concentration is going down, the teachers have to find the root causes for this symptom, and find a way to deal with it. Depending on the child and his or her needs, different causes can lead to the same symptoms. For example, learning problems may indicate family or physical issues. It is the teacher's job to find countermeasures which will help the child to feel better and more confident again. These countermeasures differ from child to child, and a great deal of experience is required to find a the best one.

Discussion: In this scenario, the parallels to the nursing domain and to FIPS in general were obvious: Just as patients who have their individual problems, children with special needs also require attention for their individual needs. If a child shows some symptoms (as described above) it is the teacher's job to identify the problem and to find an effective way of dealing with it. If the teacher is experienced in his field of work, he had similar cases before, and uses this knowledge in order to find the best way to help the child.

2.2.4 Expert Interview: Doctoral Program Supervision

We interviewed two participants in the doctoral program of the University of Vienna. During the interviews it became clear that the doctoral program can also be seen as a FIPS. While all students have their individual project, there are some common characteristics between the different students and projects which can be utilized to analyze problems and find solutions. The basic research plan is typically laid out during the first year of the PhD studies, but it has to be adapted many times during the course of the work, whenever issues or new ideas come up. For example, if a student realizes he or she requires some special equipment which is hard to get, he or she may have to adapt his or her work plan. There might be different solutions for the same challenge. For example, if a certain kind of technical equipment, like a special microscope, is required to complete a part of the work, for some projects it may be it easier to adapt the work in a way that this equipment is no longer needed. For other projects, the students may have contacts to institutions which can provide such a microscope.

Discussion: During the interviews it became clear that the doctoral program can also be seen as a FIPS. In contrast to other domains however, each student only has one individual project (i.e. one subject in terms of a FIPS) he is working with. While in the other three domains people were identifying and handling problems for several different subjects (e.g. guests, support cases, children) each student only deals with one subject.

Conclusion: Altogether 8 experts from 4 different domains were interviewed. All of these experts were able to identify some commonalities between the domains they are working in, and a FIPS. Together with the domain experts, we identified the basic building blocks of a FIPS which we discussed in the last section. For each of the four domains described above, these building blocks have been discussed in detail and allocated with data from each domain.

In addition to the building blocks which are based on related work, and the *bonus list* identified during the expert interviews, the following special properties of a FIPS were encountered while analyzing these basic building blocks with the domain experts (c.f. Section 2.2.) The following list defines these properties:

- **Individualism:** Each subject, and each situation where a trigger occurs, has its very individual properties. For example, in the software development domain, each support case has its own set of data elements which describe the situation and the case itself.
- **Limitation of available resources:** In each setting, the resources which can be used to solve the problematic situation are limited. This usually leads to a prioritization of the problems: For those which are more important, more resources are spent. In the software domain, a crucial support case naturally receives more resources in order to solve it. On the other hand, in the special needs school setting, such a prioritization is not that simple: Every child needs attention, and the available resources have to be split up in a good way.

- **Ambiguity of triggers:** The same trigger does not have to mean that the same problem has to be solved: In the special needs school, two children can show the same symptoms (e.g. they cannot concentrate any more), but the reasons behind those symptoms can be very different.
- **Diversity of possible solutions:** If we know what the actual problem is, still multiple solutions are possible. It highly depends on the current situation which one is chosen, i.e. the workload of the available resources, the history of the individual etc. In the special needs school it depends to a big part on the condition of the child which solutions can be chosen in order to make him or her feel better.

3 Designing a Tower Defense Game as an Experimentation Service for FIPS

The main goal of this work is to design a game-based experimentation service that reflects the building blocks introduced in Section 2 and enables the simulation of FIPS in a generalized way (→ Q2, Q3). Specifically, the service enables the collection of data that is essential for user support features [Gü08; KR15; We09], i.e., process execution and change logs.

The following section first discusses why we decided to implement a tower defense game as an experimentation service for FIPS. Following, we introduce the basic ideas and gameplay. After that it is explained how this gameplay relates to FIPS building blocks and procedures. Finally, the basic architecture are shown and we introduce the data players generate while playing the game.

3.1 Using a Tower Defense Game as an Experimentation Service

Tower defense games (cf. e.g. [Av11]) are strategy games where the player has to defend a set of towers. Usually, the player has some defense systems to choose from (build a cannon, train some soldiers, etc.) and sends them to the towers. In certain intervals, waves of enemies attack these towers and try to destroy them. By placing his defense systems, the player tries to counter the enemy forces and avoid his towers from being destroyed.

We chose a tower defense game as a basis for our evaluation service for several reasons: First, tower defense games are a well known type of game, thus, the basic game mechanics will be quite easy to understand for new players. Second, tower defense games can be round based. This means that the player can take as much time as he requires to make his decisions. This stands in contrast to real-time strategy games, where the player has to make his decisions very fast in order to keep up with the game. A very popular representative of a round based strategy game is chess: Here, both players can usually take a long time to decide what to do next. Since the goal of our game is to represent a FIPS, we decided to implement a round based strategy game: As in a real FIPS, where the responsible actors have to plan their next steps, the player should have enough time to plan what to do next. Third, a tower defense game does not depend on a real domain: For most FIPS, the person responsible for deciding

what to do next has to be a domain expert. Depending on the domain, it can take a long time to acquire the required knowledge. In a domain independent setting such as a tower defense game, no long training is required: Of course, it takes some time for new players to know exactly when to use which defense system, but in contrast to learning the details of nursing science, this can be done very fast. Fourth, the domain independence also leads to the advantage that anyone interested can download and play our game. Thus, we have a broad base of possible players who generate the data we can use for further evaluation.

Aside from the players who actually play the game, our targeted audience are scientists: First, the generated data can be used to develop and evaluate algorithms work with FIPS data. Second, using our game as an experimentation service, scientists can evaluate approaches to support users in FIPS domain independently. Imagine a service which aims at supporting FIPS users. Our game service could be utilized to compare two groups of FIPS users: Those who receive some kind of support when making decisions, and those who do not receive any support. If the supported group outperforms the group who did not receive support, it can be a hint that the support service might be useful. Due to the domain independence of our game service, a broad range of users can be included in such an experiment.

3.2 Gameplay

The idea of the game to be developed[4] is that the player slips into the role of a commander who has to conquer and defend villages on an island. (The villages are equivalent to the towers in the definition given above.) Each of these villages has its own set of parameters which make them individual. While some villages are closer to the sea, others are in the middle of the woods, and others close to mountains. These villages are attacked by enemy armies in varying intervals. Who these armies are and how they attack largely depends on the parameters and the history of the villages. Some enemies will preferably attack villages closer to the sea, while other enemies are more often seen in the woods and will preferably plan their attacks there.

The game is round based, meaning that the player can take as much time for his decisions as he needs. In the game each round is referred to as a day. In order to defend his villages, the player can plan the defenses for the days to come. For each village, there exists exactly one defense plan, which is implemented as a highly flexible and individual process instance. In this process instance, all the actions which have to be carried out in order to defend the village are defined. On each day, the same things happen:

1. New enemies show up at some of the villages.
2. The players defenses which are already in place (because they have been planned on previous days) fight the enemies. If the enemy wins, the village loses population. If the population reaches zero, the village is lost.
3. The player can plan new defenses for the following days for each of his villages.

[4] Available at: http://cs.univie.ac.at/project/apes

The enemy armies attack the villages over multiple days. On the first day, they usually show up with light units who do not do much damage, but the player already notices them. The longer the player does not react properly to the threat, the stronger the enemy army gets; ultimately destroying the village.

The player controls four different buildings which cannot be attacked by an enemy directly. Their sole purpose is to produce different parts relevant for defending the villages. All buildings have a maximum number of items they can produce on a single day. If their workload is fully used, they cannot produce any more on that day.

- **Barracks:** In the barracks, warriors of different types are trained. Among others, the player can choose between a knight with a lot of health points, an archer who does additional ranged damage, a mage who is a proficient spellcaster, and a soldier who does melee damage.
- **Forge:** A warrior cannot do much without a weapon. In the forge, the player can create multiple weapons such as swords, maces, axes, polearms, longbows etc. Each of these weapons has individual properties, which make them more or less effective depending on the warrior who carries them.
- **Mage Tower:** In the mage tower the player can create spells which are cast by the defense units either to protect themselves or to harm their enemies.
- **Alchemy Lab:** In the alchemy lab the player can prepare oils which add an additional effect to the weapons, and thus enhance their effectivity.

Each time a new enemy army shows up at a village for which the player has not planned any defenses yet, he is supposed to do the following steps:

1. **Choose Warriors:** In a first step, the player chooses the warriors which make up his army. These warriors are from one of three different races (human, elf and dwarf) which all have advantages and disadvantages depending on the terrain of the village and the enemy they are facing. The player can choose as many warriors as he wants for each day, as long as the total number of warriors doesn't exceed the maximum workload for the barracks for this day.
2. **Prepare the Army:** Before sending his army into battle, the player has to allocate the weapons and spells to his warriors. Basically, all infantry can carry all kinds of weapons and spells, but some will perform better with certain types of weapons than others. For example, an archer will do more damage using a longbow than a mage would do with the same weapon. Also, the effectiveness of the weapons depends on the enemy the player is facing: Some weapons will perform good against certain races, while others will not yield good results.
3. **Execute Adaptation:** When the player has set up his army for the following days, he adapts the village's process instance and sends his troops to the village which is attacked.

4. **Evaluate Results:** Each day, the enemy armies will first fight the defending armies and - if they should destroy the defenses - inflict a certain amount of damage to the village by killing its population. If the village's population reaches zero, the village is lost for the remainder of the game. If the player has lost all his villages, the game is over.

Fig. 1 shows the interface for choosing the troops and units for the defense of a village called Merasus in the game. The player can select for each day which soldiers, weapons, and defense systems he wants to use for his defense.

Magical Alignment: The spells and oils which can be created in the alchemy lab have a certain magical alignment. There are three axes of magical alignment: **Heat vs Cold**, **Order vs Chaos**, and **Nature vs Corruption**. Aside from the topology of a village, magical alignment is the second concept which creates individual situations. Each time a spell of a certain alignment is being cast at a village (either in order to defend or attack the village), the alignment of this village shifts into the direction of the spell's alignment. For example, if the player decides to send units who cast the spell *Fireball* to the battlefield (which is a *Heat* spell), the magical alignment of said village will slowly change to the *Heat* alignment, and away from the *Cold* alignment. Fig. 1 shows an overview over the magical alignment of a village. The magical alignment of a village influences how good or bad a spell or an oil of a certain alignment works. If the *Heat vs Cold* alignment of a village is more on the heat side, heat spells will generally be more effective, and cold spells won't work as well. Spells from the other alignments such as *Nature*, *Corruption*, *Order* or *Chaos* are not affected. Magical alignment adds a great deal of individualism to a village.

Fig. 1: Choosing the defense mechanisms; framed in blue: magical alignment

3.3 Mapping the Tower Defense Game Concepts to FIPS Buidling Blocks

In the following we describe how the gameplay is mapped onto FIPS.

- **Subjects** map to villages which differ according to their topology, magical alignment and their history of enemy attacks.
- A **process instance** maps onto exactly one defense plan for each village which should be adapted every time a new enemy army shows up.
- **Organizational units** map onto buildings under the player's command that carry out the steps defined in the process instance, namely the barracks, the forge, the alchemy lab and the mage tower.
- An **environment** subsumes all resources and organizational units that have to be shared for the execution of a certain process, for example, a nursing home. In the tower defense game, all production buildings have to be shared for the set of villages in the game. Hence, each game forms its own environment.
- **Process fragments** map onto the defense plans a player creates in order to stop the enemy army. These defense plans / process fragments are inserted into the village's defense process instance.
- **Problem list:** For each fragment which can be inserted into a process instance there exist some situations where it will not be optimal to execute its steps, for example if the magical alignment of the village is contradicting the spells' alignment.
- **Bonus list:** While some spells may be problematic in some situations, they may perform well in others. If a village's magical alignment is the same as the spells in a fragment, carrying out the adaptation may yield more promising results.
- **Triggers:** Every time a new enemy army shows up at a village, its defense process instance should be adapted.
- **Positive goals:** If the enemy is destroyed or chased away, the battle is won.
- **Negative goals:** If the village loses population or too many resources have been used, the battle may not have found its optimal outcome.

- **Individualism:** Each village has its own, individual properies. This includes the terrain, number of rivers, and magical alignment.
- **Limitation of available resources:** The buildings can only produce a limited amount of soldiers and weapons to defeat the enemies each day. Their resources have to be split up among all villages.
- **Ambiguity of triggers:** When a certain enemy shows up, it does not necessarily mean that always the same kind of enemy will follow. For example, if some troll scouts show up at a village, it does not always mean that the same troll army is about to attack. The players will have to take a look at the village's topological parameters, magical

alignment and history to find out which army they may be facing. Additionally, the player can send some scouts to find out more about the enemy.

- **Diversity of possible solutions:** Each enemy can be faced with many diverse defense tactics. Choosing the optimal one depends on the current situation of the village and the workload of the environment. Depending on the magical alignment of a village and the enemy army, different solutions may perform better or worse.

3.4 Comparing the game's adaption planning procedure to a real world FIPS

In Section 3.2 we presented the basic procedure of planning a defense strategy in our game. We showed what a player has to do as soon as he finds out that an enemy is attacking one of his villages, and how his plan is executed. In this section, we show how this basic procedure of planning a defense strategy for a certain village is similar to planning the upcoming actions in a real world FIPS. Table 1 shows the mapping between planning upcoming actions in a FIPS and in the game. This is shown by an example from the special needs school and compared to the actions a player has to take in the game.

Tab. 1: Mapping between the Tower Defense Game and a real world FIPS

Step	Special Needs School	Game Setting
Step 1: Trigger	teacher notices problems with a student's behavior	player sees attack of the village by some enemy
Step 2: Adaptions	teacher plans what to do to help his student	player plans the defenses for the next days
Step 3: Execution	teacher executes his plan	buildings send the planned units into battle
Step 4: Evaluation	teacher evaluates whether the problems have been solved or not	player gets feedback about the result of his defense

First, some problem has to be identified. In the special needs school, this is done by the teacher: Whenever he discovers some problems with one of his students, he has to do something about it. In our game, this is implemented as the arrival of a new army: The player knows he has to plan some kind of defense in order to defeat it. Following this initial analysis, some adaptions have to be planned: The teacher has to prepare some kind of support for the child, and the player has to plan his defense strategy. Third, the plans are executed, followed by the evaluation: In the special needs school, the teacher has to find out whether his plan has worked and his student's problems are solved or not. In the game setting, the player immediately sees whether the enemy has been defeated or not.

3.5 Service Based Architecture

The game architecture shown in Fig. 2 consists of several independent services which interact via RESTful interfaces. It consists of a *game server*, which provides the calculations for the fights and when which enemy army shows up where, a *process engine* which manages

the defense process instances for the villages, a *logging service* which logs all information into the *data repositories* and a *game interface*. In the following, we will describe each service in detail.

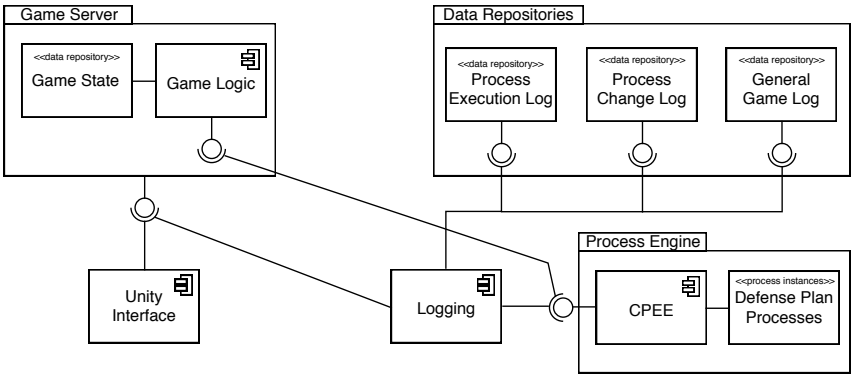


Fig. 2: Service Components and their Interactions

The Cloud Process Execution Engine (CPEE) [MR14] is a service-oriented process execution engine which itself is a RESTful web service. It manages the defense process instances and provides multiple interfaces for interacting with the process instance and logging various kinds of information.

The game server provides all calculations for the fights and the villages and stores all relevant information for the games which do not directly belong to the defense process instances. This includes among others the population, magical alignment and topography of the villages. It also communicates with the CPEE when the player updates the defense plan of a village and controls the enemy: Each round it calculates which villages of a player will be attacked by which enemy, and calculates the damage these enemies do.

The logging service generates the change and execution logs[5] for the defense plan processes which are provided by the CPEE. Additionally, it logs relevant information from the game server, which can be used to further understand and evaluate these change and execution logs. For example, the logging service logs when which enemy arrives at a certain village, how many units were lost during the fights, how the magical alignment shifts etc. In combination with the change and execution logs, the logging service provides a complete picture of each situation, which then can be used for further studies.

The user interface has been developed in Unity. It accesses the data and services provided by the game server web service over RESTful interfaces.

[5] Log format: XES, cf. http://www.xes-standard.org/

3.6 Generated Log Files

While players play the game, three types of log files are generated and updated for each village: The general log, the change log and the execution log. In this section we describe these log files in detail.

In the **general log** all information, ranging from the arrival of new enemies, over which defense units have been added, up to magical alignment shifts are shown. This log file sums up the development of a certain individual - on the one hand contingent on the decisions the player has made, on the other hand contingend on internal factors, on which the player has no influence (such as new enemy armies which arrive, population lost etc.). In our nursing example, this log file would contain some data about the development of the patient, maybe including data like his body temperature, weight, blood sugar level, or other parameters which are relevant for his treatment. Listing 1 shows an example general log of the village:

List. 1: General log example

```
<?xml version="1.0"?>
<log>
  <event id="1" type="started" day="" text="Village founded on day "/>
  <event id="2" type="newday" day="2" text="New day: 2"/>
  <event id="3" type="incoming" army="3" day="2" text="Army 3 incoming on day 2"/>
  <event id="4" type="alignmentshift" day="2" alignment="heat" shift="10" newvalue="70"/>
  <event id="5" type="defense" day="2" defensetype="weapon" defensename="Longbow" building="forge"/>
  <event id="6" type="defense" day="2" defensetype="weapon" defensename="Round Shield" building="forge"/>
  <event id="7" type="defense" day="2" defensetype="weapon" defensename="One Handed Sword" building="forge"/>
  <event id="8" type="defense" day="2" defensetype="unit" defensename="Knight" building="barracks"/>
</log>
```

The **change log** shows the process change operations the player has conducted while planning his defenses. These change operations represent the next steps which will be taken in order to save the village from the threat. Coming back to our nursing example, this log file represents the planned therapy steps which will be taken in the future in order to support the patient in dealing with a certain condition. Listing 2 shows an example for a change log from the game as it has been logged by the logging service. Here, the change shows the insertion of a new defense strategy for a specific village.

List. 2: Change log example

```
<?xml version="1.0"?>
<log xes.version="2.0" xes.features="arbitrary-depth">
  <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext"/>
  <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"/>
  <extension name="Change" prefix="change" uri="http://leonardo.wst.univie.ac.at/acaplan/change.xesext"/>
  <global scope="trace">
    <string key="concept:name" value=""/>
  </global>
  <classifier name="Change" keys="change:type change:subject"/>
  <trace>
    <string key="concept:name" value="Change Log of Village 1"/>
    <event>
      <date key="time:timestamp" value="2016-05-11 17:36:09 +0200"/>
      <date key="day" value="2"/>
      <int key="change:transaction" value="0"/>
      <string key="change:type" value="insert"/>
      <string key="change:position" value="root"/>
      <string key="change:fragment" value="17"/>
      <string key="change:rationale" value="troll scouts"/>
      <string key="change:goal" value="defeat army"/>
    </event>
  </trace>
</log>
```

The **execution log** sums up the process steps which have been executed in order to help the village to overcome the enemy army. These steps have been planned beforehand (as can

be seen in the change log). In the nursing domain, there has to exist an execution log for each patient due to legal obligations: For each patient, each therapy step which has been executed has to be strictly logged in order to trace back any errors which have may been done if something goes wrong. Listing 3 depicts an execution log that reflects how the units are added to the villages defense.

List. 3: Execution log example

```
<?xml version="1.0"?>
<log xes.version="2.0" xes.features="arbitrary-depth">
  <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext"/>
  <trace>
    <event>
      <int key="day" value="1"/>
      <string key="concept:name" value="archer trained"/>
      <string key="unit" value="archer"/>
      <string key="building" value="barracks"/>
    </event>
    <event>
      <int key="day" value="1"/>
      <string key="concept:name" value="knight trained"/>
      <string key="unit" value="knight"/>
      <string key="building" value="barracks"/>
    </event>
    <event>
      <int key="day" value="1"/>
      <string key="concept:name" value="fireball created"/>
      <string key="spell" value="fireball"/>
      <string key="building" value="alchemylab"/>
    </event>
  </trace>
</log>
```

These logs provide valuable data sources for deriving insights on the effects of previously applied changes based on process analysis techniques. In this section we have shown some parallels between our log files and the nursing domain. In the next section, we will evaluate the log files with equivalent data from the software sales and support domain.

4 Evaluation

The goal of the game-based setting is to provide log data that are not subject to any disclosure restrictions and a test setting which can be used to develop algorithms and approaches which support users when adapting process instances in a FIPS. The different types of log files, namely change, execution and general logs, have been introduced in the last section. In this section we want to show that the information gathered in these log files are actually comparable to the set of information which is gathered in a real world FIPS. We claim that if the data generated in our game matches the data generated during an individual situation in a real world FIPS on a conceptual level, our data can be used to evaluate approaches and algorithms to support process adaptions in FIPS.

For our evaluation we analyzed support cases from mesonic[6], the ERP / CRM software developer introduced in the expert interviews in Section 2.2.1. In their *support network*, mesonic has over 200.000 different support cases describing the respective problem in free text form. Each support case is classified in one of four groups, ranging from bugs, where the

[6] Due to data privacy issues we only publish anonymized parts of the actual data gathered from these support cases in this paper. Since the original support cases are in German free text form, we shortened them and translated the relevant parts.

development team has some work to do, over wishes, where it has yet to be decided whether the desired feature will be implemented or not, over general support cases up to questions which are already answered in the manual. Depending on the case's classification, the next steps are decided: If it is a real bug or a wish, it will be discussed with the responsible developer or, if necessary, the whole team. If it is a general support case, it will be handeled in the support department internally. If it is something which can also be found in the manual, the responsible supporter handles the case on his own. This classification represents the first steps in the planning stage: Based on a cases "symptoms", the next steps are planned. In addition to this classification, information such as the customer's name and id, the retailers name and id, the version and build number of the product are logged.

Figure 3 summarizes the comparison of the data elements in the real world FIPS with the data elements in the game: The top section shows an example of such a support case: Here, some problems with the OLAP component have appeared which have been classified as a bug. On the bottom, example log files from the game are shown. The mappings between the data elements are indicated by the arrows. The general data of the support case (i.e. customer id, retailer id etc.) is not part of any log file, but static information which is saved separately.

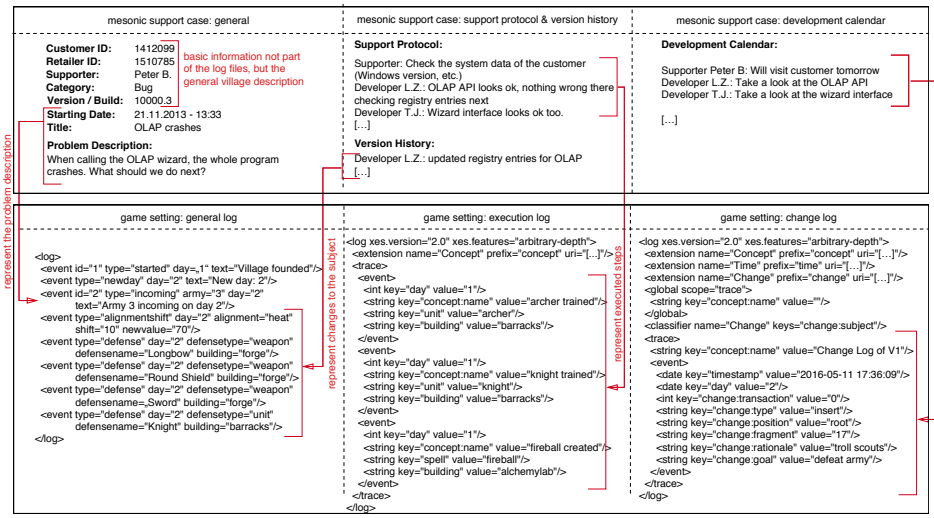


Fig. 3: Mapping of the log files to a real world FIPS scenario

Data Source Comparison 1: The change log: At mesonic, next steps are planned with the responsible employees and shown in their calendars. In the game, the change log shows the planned steps for solving a village's problem (i.e. the incoming enemy army).

After the next steps for handling the case are planned, they are executed by the responsible employees. The executed steps are documented in the support case itself in free text format.

Data Source Comparison 2: The execution log: Mesonic's support case documents what has been done, who has done it, and when it has been done in order to solve the problem

at hand. The execution log from our game represents the very same type of information: It is shown when which step has been executed in order to deal with the village's problem. This information includes the time, what has been done, and which role has executed the steps.

All actions influence the state of the product. These state changes are documented in the support protocol as well as the company's version control system.

Data Source Comparison 3: The general log: The version history of mesonic logs all information about changes to the product's source code. These changes may be because of actions which have been applied because of planned steps within the process, or factors not related to the support case. In the game, the general log summarizes all state changes of a village's parameters. These changes happen because of the player's actions, or because of other events which are not influenced directly by the player (but e.g., by the enemy army).

Conclusion: The comparison between the three generated log files from the game and the data gathered in a real world FIPS, namely the software sales and support setting, shows that all relevant information to a FIPS can be found in the game logs. In the real world setting, however, many data sets are free text, thus making further analysis difficult. Here we see a big potential of the game: The structured data in XES respectively XML can be easily used for further analysis, while representing all information relevant to a real world scenario.

5 Related Work

Virtual settings have already been used for process modeling. In [BRW11] the authors have created a 3D virtual world to enhance the collaborative modeling of business processes - even if the participants are not in geographical vicinity. By using avatars - 3D representations of humans who engage in the modeling of business processes, this approach offers various forms of communication which would be typically only possible if the participants were in geographical vicinity, i.e. speech, artifacts, gestures etc. [Ha16] presents another virtual world approach for generating process models. In contrast to other approaches, participants do not have to know any modeling language in order to generate a process model. Instead, they behave as they would do in the real world. Based on their actions in a realistic environment, a process model is generated.

While developing the game-based environment, we had to design the game mechanics according to the FIPS requirements, while still making the game fun to play. This balancing act is also an issue in serious games such as educational games [WK11], where fun game mechanics have to be balanced with a serious background [Mo08].

Data obtained from computer games, especially from strategy games, has been used for the development and validation of artificial intelligence approaches on multiple occasions. In [Av11], the authors show that tower defense games provide a good test environment for numerous computational intelligence scenarios, ranging from map generation, over enemy strategies, good defense strategies up to dynamic game balancing which keeps the player

engaged and the enemies challenging. In [On13], the authors provide an overview over the possibilities of artificial intelligence in a real time strategy environment.

There exists a multitude of approaches on flexible process management [RW12]. First requirements for FIPS have been introduced in [Ka14]. This paper extends these requirements to a comprehensive list of building blocks for FIPS. Moreover, to the best of our knowledge, there is no approach that maps building blocks of FIPS to a game-based experimentation service. Logs produced by the experimentation service can be utilized by existing approaches on user support in changing business processes such as [Aa09; Gü08; KR15; We09]. However, developing user support techniques are outside the scope of this work. Adaptive Case Management (ACM) [MS13] also handles individual cases which evolve over time. In contrast to FIPS, where a process instance evolves around a certain subject and its environment, in ACM each case is handled individually.

6 Conclusion

FIPS can be found in many different domains, ranging from the nursing sector, over hotel guest and software customer interaction to special needs schools and the PhD program. Since all of these domains deal with highly sensitive data, it is almost impossible to get a full picture of a situation where such a process instance has to be adapted without running into data privacy and legal issues. For this reason we have developed a tower defense game where the players are put into a comparable situation: They adapt highly individual process instances in order to deal with problematic situations. The data thus generated can be used without any issues, may they be legal or otherwise. Additionally, the setting itself can be used as an evaluation method for support methodologies: A set of players who received support while adapting their process instances can be compared with a set of players who did not receive any support. If the supported group of players performs way better, it may be an indicator that the support is actually helpful. In the future we will use the game in order to develop and enhance support and analytical capabilities for FIPS. This includes analyzing change logs, finding out when the desired goals are reached, and, ultimately, finding the best adaptation for a given situation.

Acknowledgment This research has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT15-072.

References

[Aa09] van der Aalst, W. et al.: Declarative workflows: Balancing between flexibility and support. Computer Science-Research and Development 23/2, pp. 99–113, 2009.

[Av11] Avery, P.; Togelius, J.; Alistar, E.; van Leeuwen, R. P.: Computational intelligence and tower defence games. In: 2011 IEEE Congress of Evolutionary Computation (CEC). Pp. 1084–1091, June 2011.

[BRW11] Brown, R.; Recker, J.; West, S.: Using virtual worlds for collaborative business process modeling. Business Process Management Journal 17/3, pp. 546–564, 2011.

[Gü08] Günther, C.; Rinderle-Ma, S.; Reichert, M.; van Der Aalst, W.; Recker, J.: Using process mining to learn from process changes in evolutionary systems. International Journal of Business Process Integration and Management 3/1, pp. 61–78, 2008.

[Ha16] Harman, J.; Brown, R.; Johnson, D.; Rinderle-Ma, S.; Kannengiesser, U.: Augmenting process elicitation with visual priming: An empirical exploration of user behaviour and modelling outcomes. Information Systems/, 2016.

[Ka14] Kaes, G.; Rinderle-Ma, S.; Vigne, R.; Mangler, J.: Flexibility Requirements in Real-World Process Scenarios and Prototypical Realization in the Care Domain. In: OTM 2014 Workshops. Pp. 55–64, 2014.

[KK97] Kueng, P.; Kawalek, P.: Goal-based business process models: creation and evaluation. Business Process Management Journal 3/1, pp. 17–38, 1997.

[KR15] Kaes, G.; Rinderle-Ma, S.: Mining and Querying Process Change Information Based on Change Trees. In: Service-Oriented Computing. Pp. 269–284, 2015.

[Mo08] Moreno-Ger, P.; Burgos, D.; Martínez-Ortiz, I.; Sierra, J. L.; Fernández-Manjón, B.: Educational game design for online education. Computers in Human Behavior 24/6, pp. 2530–2540, 2008.

[MR14] Mangler, J.; Rinderle-Ma, S.: CPEE - Cloud Process Exection Engine. In: Int'l Conference on Business Process Management. CEUR-WS.org, Sept. 2014.

[MS13] Motahari-Nezhad, H. R.; Swenson, K. D.: Adaptive Case Management: Overview and Research Challenges. In: 2013 IEEE 15th Conference on Business Informatics. Pp. 264–269, July 2013.

[On13] Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; Preuss, M.: A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. Computational Intelligence and AI in Games 5/4, pp. 293–311, Dec. 2013, ISSN: 1943-068X.

[RW12] Reichert, M.; Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer, 2012.

[Sc08] Schonenberg, H.; Mans, R.; Russell, N.; Mulyar, N.; van der Aalst, W.: Process flexibility: A survey of contemporary approaches. In: Advances in Enterprise Engineering I. Springer, 2008, pp. 16–30.

[We09] Weber, B.; Reichert, M.; Rinderle-Ma, S.; Wild, W.: Providing Integrated Life Cycle Support in Process-Aware Information Systems. International Journal of Cooperative Information Systems 18/01, pp. 115–165, 2009.

[WK11] Wallner, G.; Kriglstein, S.: Design and Evaluation of the Educational Game DOGeometry: A Case Study. In. ACE '11, ACM, Lisbon, Portugal, 14:1–14:8, 2011, ISBN: 978-1-4503-0827-4.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 351

An Experimental Analysis of Different Key-Value Stores and Relational Databases

David Gembalczyk[1], Felix Martin Schuhknecht[2], Jens Dittrich[3]

Abstract: Nowadays, databases serve two main workloads: Online Transaction Processing (OLTP) and Online Analytic Processing (OLAP). For decades, relational databases dominated both areas. With the hype on NoSQL databases, the picture has changed. Initially designed as inter-process hash tables handling OLTP requested, some key-value store vendors have started to tackle the area of OLAP as well. Therefore, in this performance study, we compare the relational databases PostgreSQL, MonetDB, and HyPer with the key-value stores Redis and Aerospike in their write, read, and analytical capabilities. Based on the results, we investigate the reasons of the database's respective advantages and disadvantages.

Keywords: Relational Systems, Key-Value Stores, OLTP, OLAP, NoSQL, Experiments & Analysis

1 Introduction

Nowadays, databases are almost present everywhere. Obvious application areas are for instance Big Data Analytics, back-ends for e-commerce systems, session storage for webservers, or embedded systems like smartphones or activity trackers. Although databases are applied in so many places, they serve mainly two workloads: Online Transaction Processing (OLTP) and Online Analytic Processing (OLAP). Relational databases have started their triumphant advance after the introduction of the relational model in the area of the databases in 1970 [Co70]. Some of them focused only on OLAP or only OLTP deploying specific optimizations, respectively. Since around 2009, a different category of databases appeared and became hyped, the so-called NoSQL databases [SF12, pp. 9-12]. Within this category, one group of competitors for the domain of OLTP are key-value stores. Initially intended to serve just as inter-process hash tables, they provide today much more functionality than just storage and retrieval of key-value pairs. Some vendors, such as Aerospike, even started to tackle the area of OLAP.

Regarding the recent development, the following question emerges: what distinguishes these systems besides the way of storing their content and what are their advantages and disadvantages? To provide an answer to this question, we compare PostgreSQL, MonetDB, and HyPer as relational databases and Redis and Aerospike as key-value stores. Additionally, we use two data types, Hstore and JSONB, within PostgreSQL to simulate a storage similar to Redis and Aerospike in order to get a better understanding of how key-value stores differ from classical relational systems. We make a performance study to compare the aforementioned databases and simulations in three categories: write queries, read queries,

[1] Saarland Informatics Campus, Information Systems, E1.1 66123 SB, s9dagemb@stud.uni-saarland.de
[2] Saarland Informatics Campus, Information Systems, E1.1 66123 SB, felix.schuhknecht@infosys.uni-saarland.de
[3] Saarland Informatics Campus, Information Systems, E1.1 66123 SB, jens.dittrich@infosys.uni-saarland.de

and more complex analytical queries. In the first category, we investigate how fast these databases can insert and delete content. The second category focuses on two access methods to read content: besides of simple selects, we examine the efficiency of secondary indexes. Finally, the last category utilizes queries which are provided by TPC-H [Co14] and compares the databases' OLAP capabilities.

1.1 Relational Databases and Key-Value Stores

Table 1 and Table 2 show the relational systems respectively the key-value store we discuss in this work. Alongside, we present the configurations that have been applied in this evaluation. Note that the used configurations are only subsets of all possible configurations.

Configuration	**PostgreSQL**	**MonetDB**	**HyPer** (Demo) [KN10]
Layouts	Row-store, Hstore, JSONB	Column-store	Column-store
Concurrency	MVCC	Optimistic CC	Chunking & MVCC
Secondary Indexes	yes (manual)	yes (on-demand)	yes (manual)
Prepared Statements	yes	yes	no

Tab. 1: **Relational Systems** alongside with their properties.

Configuration	**Redis**	**Aerospike**
Layouts	Predefined Datastructures (Hashes, Ordered Sets)	JSON
Concurrency	Serialized	Multi-threaded
Secondary Indexes	simulated with Ordered Sets	yes (manual)
UDFs	yes (LUA)	yes (LUA)

Tab. 2: **Key-Value Stores** alongside with their used properties.

1.2 Related Work

In the past years, there has been work on comparing systems of different types. In [AMH08] the authors try to answer the question how different column-stores and row-stores are. They find that column-stores can be simulated by row-stores to a certain degree and thus speed-up the queries. The authors of [Kl15] compare Riak, MongoDB, and Cassandra as representatives of the NoSQL-groups key-value store, document store, and column store respectively using the YCSB benchmark. One key point in this comparison is the influence of different consistency assumptions within a cluster of nine nodes. A performance comparison between Microsoft SQL Server Express and MongoDB is made in [PPV13] based on a custom benchmark. According to their findings MongoDB is faster with OLTP queries using the primary key whereas the relational database is better with aggregate queries and updates based on non-key attributes. In [Fl12] the authors draw a comparison between Microsoft's relational database Parallel Data Warehouse (PDW), the document store MongoDB, and the Hadoop based solution Hive. In the TPC-H benchmark PDW is for smaller data sets up to 35 times faster than Hive. For the largest set PDW is still 9 times faster. PDW and Hive are faster than both MongoDB versions (one with client-side sharding and one with server-side sharding) which "comes in contrast with the widely held belief that relational databases might be too heavyweight for this type of workload" [Fl12].

1.3 Comparing Apples and Oranges

Comparing two databases, by all means, is not an easy task and may result in comparing apples to oranges. The first and maybe most obvious point is comparing disk-based with in-memory systems. To minimize this discrepancy, we do the following: first, we store the database files on a RAM disk to improve at least the disk access times. Second, if possible we enlarge the caches and make sure the benchmark makes the same requests during each run. During the first run the database will load most of the data into the caches. Afterwards, we omit the first run from the results. Another difficulty are the different client interfaces used to communicate with the servers. SQL in connection with a programming language specific database binding is the de facto standard to use relational systems in client applications. In the case of Java it is JDBC and a database specific driver. In contrast, as a result of the variety of features, almost all key-value stores have their own client libraries, sometimes even multiple different libraries such as in the case of Redis. Finally, it remains the question how to compare single and multi-threaded databases. Well, there are multiple ways to enforce a single-threaded usage or to simulate multi-threading using multiple local instances and client-side sharding. Either way, it is unfair to one or another because they are particularly designed with one of these two concepts in mind. We will discuss these comparison issues in the respective experiments in more detail.

2 Experimental Setup

All experiments are performed on a machine equipped with two Intel Xeon X5690 hexacore CPUs running at 3.47 GHz with 192 GB DDR3-1066 RAM. All BIOS settings are set to default. In total the system runs with 24 hardware threads. The installed operating system is Debian 8.3 with kernel version 3.16.0-4-amd64 and openjdk 1.7.0 64-bit is used as java runtime environment. The following versions of the databases and their bindings are used:

- PostgreSQL 9.5.2 with PostgreSQL JDBC driver 9.4.1207
- MonetDB 11.21.13(Jul2015-SP2) with MonetDB JDBC driver 2.19
- HyPer 0.5 demo with PostgreSQL JDBC driver 9.4.1207
- Redis 3.0.6 with java client jedis 2.8.0
- Aerospike 3.7.2 community edition with java client 3.1.8

2.1 A Custom Benchmark

The simplest way would be to use and extend YCSB (Yahoo! Cloud Serving Benchmark) [Co10]. Nevertheless, we are going see in the first experiment that YCSB has one major disadvantage: it has a rather poor performance. The reason for this lies in its design decisions, which are aimed at providing flexibility and extensibility. Therefore, we create a custom benchmark tool to address these problems. A requirement besides the high throughput is scalability in terms of concurrent clients and batch sizes. While YCSB has also support for multiple clients it lacks the support for grouping multiple queries of the same type into one batch. Using batches reduces the amount of requests send to the database and in consequence overhead by network IO. As foundation for the benchmarks a slightly modified TPC-H schema is used. An additional attribute has been introduced into both the `partsupp` and `lineitem` tables which serve as artificial primary keys. This change is made rather for the relational databases than for the key-value stores. A concatenated key might result in multiple comparisons, for each part of it, whereas in key-value stores

always a single value is used as primary key. On top of it the provided generator tool is used to generate the test data. Overall the new benchmark tool is split into three main parts: (1) The query generator is responsible for creating all query data in a generic way. (2) The query converter, as the name states, converts the generic query data as far as possible into actual database specific statements. Afterwards, these are stored in a shared queue. (3) The actual query threads get the statements out of the shared queue and perform the queries. Besides, in case of PostgreSQL and MonetDB we use prepared statements. The available HyPer demo lacks of support for prepared statements. Therefore, usual statements are used.

2.2 Database Schema

In general, seven database variants are subjects in the following experiments: the relational ones are PostgreSQL, HyPer, and MonetDB which are respectively denoted as **PG-row**, **HyPer**, and **MonetDB**. In addition, for PostgreSQL we also test the Hstore and JSONB data types denoted as **PG-hstore** and **PG-jsonb**. With these two data types we are able to simulate the behavior of Redis and Aerospike with PostgreSQL. The key-value stores Aerospike and Redis are denoted as **AS-simple** and **Redis**. As mentioned previously a modified TPC-H schema is used for the benchmarks. It is obvious that this schema is replicated inside the relational databases. Nevertheless, the schema needs to be mapped to both key-value stores, PG-hstore, and PG-jsonb. For Redis all records are stored in one table. The keys consist of the table name and the primary key value, for example `nation3`, `customer146`, or `lineitem5890412`. Hashes are used as data structures for the values with the column names as the corresponding field names. For Aerospike each table goes into a corresponding set and the primary keys of the rows serve as the keys for the records within the sets. The columns of each table are also mapped to corresponding bins in each record. For PG-hstore and PG-jsonb a similar schema with all eight tables is used but all tables have only two columns. One is the primary key and the other is the value, which takes all the attributes. All three PostgreSQL variants are stored within separate databases.

3 Experimental Analysis

Each experiment contains one or more benchmark types. For each database variant the benchmark performs four consecutive runs, one warm-up and three measuring runs. During each run, 50,000 operations are executed. In case of the read experiments, each run performs exactly the same set of operations. Before each experiment is conducted all databases are initially loaded with content from the generator tool used for TPC-H with SF 1. For MonetDB an extra warm-up run is necessary because of its caching behavior. It decides based on the query whether to create caches or not. In return, not all warm-up runs enforce the creation of caches. In this additional run 1,000,000 rows are selected using the primary key. The benchmarks are executed either for an increasing amount of concurrent clients using one single operation per request or for an increasing batch size using only one client.

3.1 Setting the Baseline

Before we can start with the full-fledged experiments that measure end-to-end runtimes, let us begin with a simple experiment, that focuses purely on the overhead of the communication with the system. Most importantly, we want to identify wether the benchmark tool itself can be part of the overhead. For the relational systems, we simply fire a `SELECT 1` query in SQL. For Redis, we can perform an echo operation. Unfortunately, this is not possible

in Aerospike, where we fire the `keyExists()` function to test for a non existing key in an empty database. Additionally, to prove whether the measured performance is bounded by the benchmark at all, the same experiment is made without communicating with any database, denoted as **NO-DB**, where simply a counter is increased while the rest remains *exactly* the same.

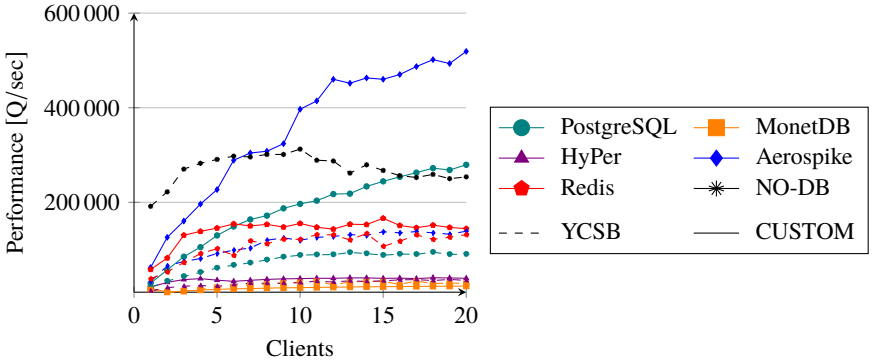


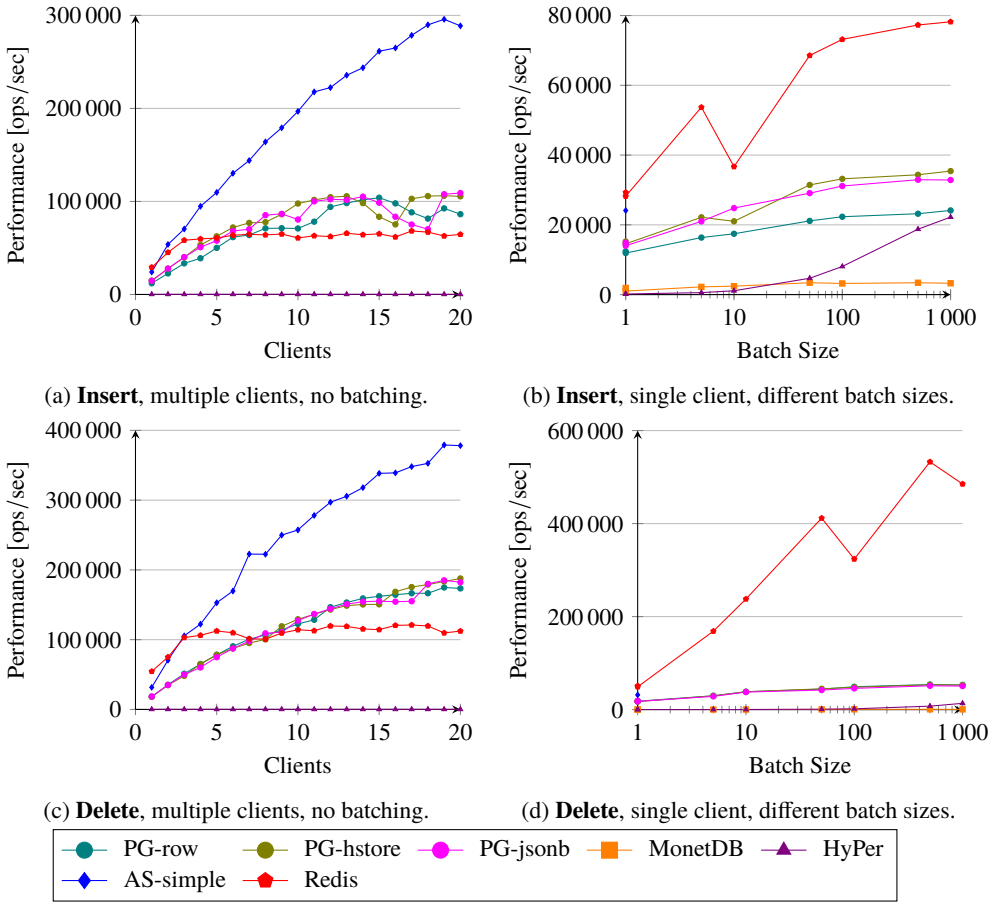
Fig. 1: **Echo Performance**. 50,000 echo requests are performed using multiple clients. **YCSB** are the results from the YCSB tool and **CUSTOM** are the results from our custom benchmark.

First, let us have a look on the results for YCSB in Figure 1. NO-DB shows the maximal throughput which YCSB is able to achieve. The results of both benchmark tools for PostgreSQL, Redis, and Aerospike show that their performance in YCSB is bounded by the benchmark itself. MonetDB and HyPer perform in both benchmarks equally. Thus, their throughput is obviously bounded by the database. As a consequence, in following experiments we are not going to take YCSB into account and consider our CUSTOM benchmark as a more meaningful alternative.

3.2 Write Experiments

In this category we start by investigating how well insertion and deletion from `orders` and `lineitem` is performed. New rows are created using the generator tool from TPC-H and adjusted to fit into our modified schema. Aerospike and MonetDB underlie some restrictions which prevent them to be part in all experiments in this section. Aerospike is not capable of batched insert or delete operations and is used only in experiments with multiple clients. Since, MonetDB uses optimistic concurrency control (OCC) for transaction management it is used only in the batched experiments because OCC prevents concurrent modifying transactions.

A side note on all batch experiments: the results in Figure 2(b) show two measured values for the batch size of one: an unbatched variant of a single operation and a batched variant containing a single operation. This is necessary because batching operations together into one transaction comes at a cost, which is the difference between both measured points. The higher value is mostly the unbatched variant, with HyPer as the only exception. The results in Figure 2(a) and 2(c) show a similar outcome as the echo experiment in the previous section. Aerospike and PostgreSQL scale along with the amount of clients. In contrast to Aerospike, PostgreSQL does not scale as good as in the echo experiment. The most likely reason is the



(a) **Insert**, multiple clients, no batching.

(b) **Insert**, single client, different batch sizes.

(c) **Delete**, multiple clients, no batching.

(d) **Delete**, single client, different batch sizes.

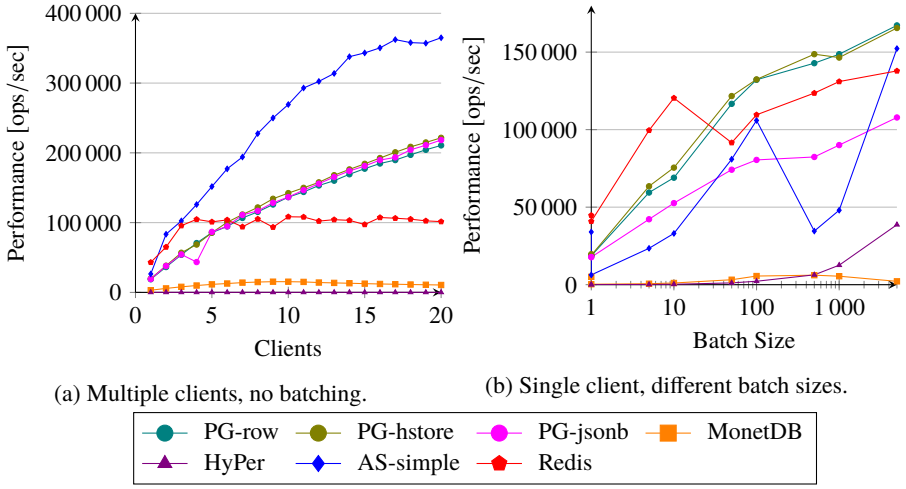
Fig. 2: **Write Performance**. In total 50,000 rows are inserted/deleted from `orders` and `lineitem`.

additional locking mechanism to provide concurrent transactions. These costs are hidden for Aerospike because they are already accounted when checking whether a record exists and thus the echo performance is reduced. Although multiple clients are used, HyPer has a bad performance. The reasons are its view on concurrent transactions in addition to the overhead induced by query compilation. As we never introduced a vertical partitioning to the schema, all transactions are serialized. A look at the CPU usage during the benchmark hardens this point as only one core is used. However, the batched experiments give us a notion of how fast compiled queries are. Among all relational databases HyPer shows the best performance improvement along increased batch sizes. From the results in 2(b), we derive two very interesting points. First, PG-hstore and PG-jsonb are faster than PG-row. While both key-value variants have to send and store additional information about the structure they need just two integrity checks: are primary keys integers and are the values of type Hstore or JSONB. Second, the single-threaded Redis instance competes with up to six processes forked by PostgreSQL. Furthermore, for a single client it is also faster than Aerospike. This may have two reasons: a faster storage-layer due to its focus on non-nested data structures or it utilizes its single thread better than Aerospike or PostgreSQL because

it does not have to lock the data against other threads. Furthermore, Redis has a downward spike in both batched experiments. This may be the result of the transaction mechanism which is used for batching the requests. Similar to transactions in relational databases, Redis stores all operations until an EXEC command is received. Upon this command, all previously stored commands are executed. Delete operations have a much smaller size and the spike apears later. Thus, this downward spike may indicate that the buffer which stores the commands is enlarged.

3.3 Read Experiments

To evaluate the read performance, let us now first look at simple selects on the primary key column. With this set of experiments we conclude the basic OLTP request types. In this section, we use all tables of our schema, that contain at least as many rows as the batch size. Furthermore, no key occurs twice within one batched request. Let us first look at Figure 3(a), where we vary the number of clients and avoid batching. We can see that Aerospike scales the best with the number of clients as one would expect from a multi-threaded key-value store under a concurrent OLTP workload. In contrast to that, Redis serializes the requests from all clients and thus saturates quickly. PostgreSQL scales less effectively than Aerospike but still improves till 20 clients. On the other end of the performance lie MonetDB and HyPer, which suffer from their focus on OLAP and expensive query compilations. In Figure 3(b), we vary the batch size while limiting the evaluation on a single client. Interestingly, all systems monotonically gain performance with an increase of the batch size except of Aerospike and Redis. Especially Aerospike suffer under the batch sizes 500 and 1000, probably due to a suboptimal distribution to the executing threads. Furthermore, we can also observe that PG-jsonb is slower than PG-row and PG-hstore with larger batches: PG-jsonb has to send significantly more meta-data.



(a) Multiple clients, no batching.

(b) Single client, different batch sizes.

Fig. 3: **Select Performance**. In total 50,000 rows are selected from all tables.

Let us now focus on secondary indexes. These are very important to answer point-queries or in join operations. Thus, we will now inspect how the systems behave when using their respective secondary index structures. The larger the table the more impact an index has on the query; therefore, we employ only the three largest tables `lineitem`, `orders`, and

`partsupp`. `l_orders`, `o_custkey`, and `ps_partkey` are used as indexed columns. We use three different query types, shown in Listing 1, that either retrieve all selected rows (**SELECT**), count all selected rows (**COUNT**), or aggregate the maximum over the selected rows of an unindexed attribute (**MAX**). In the experiments of Figure 4, we vary the size of the selected range, deactivate batching, and use eight concurrent client threads. Redis does not have secondary indexes but its developers provide an official workaround to simulate them with build-in functions. The SQL statements (see Listing 1) can be used instantly for all relational databases whereas for Aerospike and Redis these statements need to be translated. For instance, to count and to aggregate the elements, Aerospike needs to apply a stream UDF.

```
// Retrieve/Count/Aggregate records
SELECT [* | COUNT(*) | MAX([ps_supplycost|o_totalprice|l_discount]) ] FROM
    [partsupp|orders|lineitem]
WHERE [ps_partkey|o_custkey|l_orderkey] = someValue
// Example how different selectivities are realized using a range query
SELECT * FROM
    [partsupp|orders|lineitem]
WHERE start <= [ps_partkey|o_custkey|l_orderkey]
  AND [ps_partkey|o_custkey|l_orderkey] <= end
```

List. 1: All three queries to measure index performance in this experiment.

As we can see in Figure 4, for all range sizes the performance of MonetDB is almost stable, as no index is used by the system to answer the queries. Only at a range size of one, MonetDB performs better as it can test for equality. A similarly stable performance can be observed for HyPer. We can also see that PostgreSQL obviously has the best support for secondary indexes, with PG-hstore and PG-jsonb having a better performance than PG-row. This is caused by PostgreSQL building an additional bitmap index based on the already existing indexes. For Aerospike we are able to make two conclusions: first, stream UDF's have slow start times, because SELECT is faster than MAX() and COUNT(*) for the range size of one, although these two return much less data. Second, Aerospike seems to have performance issues with range queries. With larger ranges, Aerospike becomes much slower for all three queries than PG-row with SELECT, which contradicts the results of the simple selects. The simulated index for Redis shows a moderate performance.

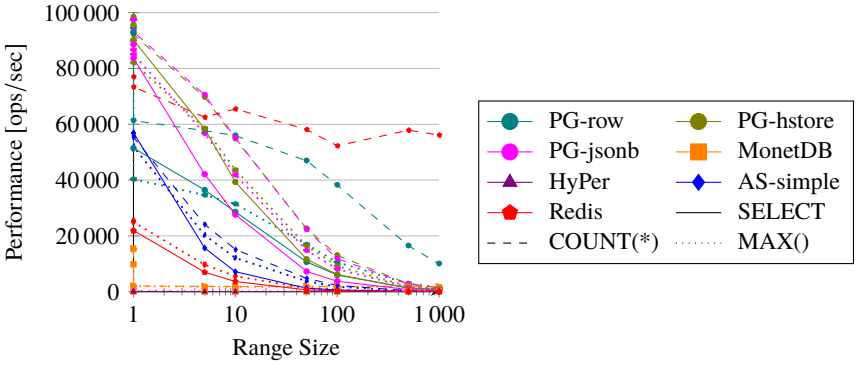


Fig. 4: **Index Performance**. In total 50,000 queries are performed using an indexed column as filter. The selectivity is represented as range of valid values.

3.4 Analytic Query Experiments

In this final category we focus on OLAP and use the queries provided by TPC-H to compare the databases in this area. Due to the lack of a join operation in Aerospike, we focus on the OLAP queries which use only one table (Q01 and Q06). In the experiments, we perform five runs with one OLAP query per run and take the average. Furthermore, we switch from the custom benchmark tool to the interface applications provided with the databases to simplify the execution, as no batching is needed anymore. For PG-hstore and PG-jsonb we map the columnar values to the attributes stored in the Hstore and JSONB values. For Redis and Aerospike, the declarative SQL statements are translated into procedural UDFs.

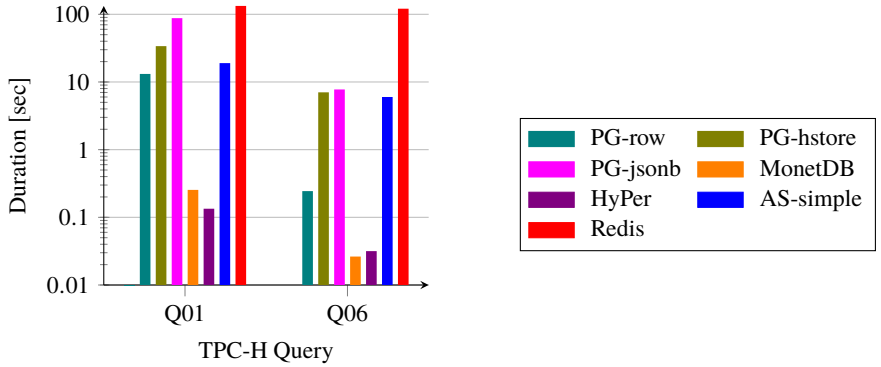


Fig. 5: **Duration of Singletable TPC-H queries.**

Let us inspect the results in Figure 5. The varying run-times for all PostgreSQL variants in Q01 present again evidence that these have very different access times to a single attribute. The most expensive access accounts for JSONB values. In all cases, the whole table is scanned with corresponding filters. Unlike in Section 3.3, this time PostgreSQL decides to perform a scan on the table in any case. Redis shows the longest run-time due to slow UDF's, similar to Section 3.3. In contrast, Aerospike shows a much better run-time which is even comparable to PG-hstore. This outcome is contrary to the findings in Section 3.3 where Aerospike is much slower than PG-hstore particularly for larger ranges. Thus, Aerospike is faster at scanning a set than querying an index. The best results yield HyPer and MonetDB. Although they use different approaches, both perform almost equally. Still, compiling the query takes some time but in the return a much faster execution is achieved because of data-centric code. Especially, MonetDB proves that its optimization towards OLAP and its custom assembly language are as efficient as query compilation.

4 Conclusion

This paper constitutes a performance study using various benchmarks[4]. Test subjects were Aerospike and Redis as key-value stores and PostgreSQL, MonetDB, and HyPer as relational databases. Additionally, PostgreSQL is used to simulate the behavior of Aerospike and Redis using the data types Hstore and JSONB. Both, HyPer and MonetDB are very

[4] Due to the page limitations, we reduced our evaluation to the presented content. The interested reader can additionally find the evaluation of deletes, joins, and multi-table TPC-H queries on our website.

efficient with OLAP queries and obtain run-times in the range of milliseconds for TPC-H queries. In return, both are not able to handle OLTP requests to a satisfying extent. For HyPer, this is a side effect of relying on query compilation. Without support for prepared statements its performance is bounded by the compilation for short requests. In contrast, the low OLTP performance by MonetDB is the result of architectural decisions. In favor of OLAP performance, the OLTP throughput is neglected. The highest OLTP throughput achieves Aerospike by utilizing all threads which are provided by the system. At the same time, the threads are used inefficiently. The reason is the locking mechanism, which is necessary to manage concurrent access to the records. Furthermore, Aerospike performed almost as good as PostgreSQL when processing OLAP requests. Due to its single-threaded design Redis has the best performance per thread. However, the downside is its inability to scale automatically along with multiple clients. Instead, the user needs to decide whether multiple Redis instances are needed or not. PostgreSQL has proven to be an all-round database throughout all tested candidates. It has the best OLAP performance behind both column-stores and the best OLTP performance behind Aerospike. Additionally, due to the Hstore and JSONB data types it can be used as key-value store. With the help of PostgreSQL we show that key-value stores have a small advantage towards OLTP requests, as schema-free databases key-value stores do not make any assumptions on the content of a value and can omit integrity checks.

Summing it up, key-value stores are particularly recommended in situations with high frequent OLTP and are not yet ready to perform well under OLAP workloads. For workloads with almost only OLAP, specialized databases are suggested. HyPer and MonetDB are just two possible candidates for such workloads. PostgreSQL provides a good trade off for mixed workloads.

References

[AMH08] Abadi, D. J.; Madden, S. R.; Hachem, N.: Column-stores vs. Row-stores: How Different Are They Really? SIGMOD '08, ACM, New York, NY, USA, pp. 967–980, 2008.

[Co70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM, 13(6):377–387, June 1970.

[Co10] Cooper, B. F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R.: Benchmarking Cloud Serving Systems with YCSB. SoCC '10, ACM, New York, NY, USA, pp. 143–154, 2010.

[Co14] TPC Benchmark H (Decision Support) Standard Specification Revision 2.17.1.

[Fl12] Floratou, A.; Teletia, N.; DeWitt, D. J.; Patel, J. M.; Zhang, D.: Can the Elephants Handle the NoSQL Onslaught? Proc. VLDB Endow., 5(12):1712–1723, August 2012.

[Kl15] Klein, J.; Gorton, I.; Ernst, N. et al.: Performance Evaluation of NoSQL Databases: A Case Study. PABS '15, ACM, New York, NY, USA, pp. 5–10, 2015.

[KN10] Kemper, A.; Neumann, T.: , HyPer - Hybrid OLTP&OLAP High Performance Database System, 2010.

[PPV13] Parker, Z.; Poe, S.; Vrbsky, S. V.: Comparing NoSQL MongoDB to an SQL DB. ACMSE '13, ACM, New York, NY, USA, pp. 5:1–5:6, 2013.

[SF12] Sadalage, P. J.; Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 1st edition, 2012.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 361

Spotlytics: How to Use Cloud Market Places for Analytics?

Tim Kraska[1] Elkhan Dadashov[1] Carsten Binnig[1]

Abstract: In contrast to fixed-priced cloud computing services, Amazon's Spot market uses a demand-driven pricing model for renting out virtual machine instances. This allows for remarkable savings when used intelligently. However, a peculiarity of Amazon's Spot market is, that machines can suddenly be taken away from the user if the price on the market increases. This can be considered as a distinct form of a machine failure. In this paper, we first analyze Amazon's current spot market rules and based on the results develop a general market model. This model is valid for Amazon's current Spot service but also many potential variations of it, as well as other cloud computing markets. Using the developed market model, we then make recommendations on how to deploy analytical systems with the following three fault-tolerance/recovery strategies: re-execution as used by traditional database systems, checkpointing as, for example, used by Hadoop, and lineage-based recovery as, for example, used by Spark. The main insights are that for traditional database systems using significantly more instances/machines can be cheaper, whereas for systems with checkpoint recovery the opposite is true, while lineage-based recovery is not beneficial for cloud markets at all.

1 Introduction

Cloud Computing has become a true utility. In particular for analytics cloud computing offers many benefits such as instant access to virtually infinite resources. However, cloud computing is not for free. Service providers such as Amazon Web Services (AWS) charge a premium that can quickly add up, even for running small analytical tasks. Yet, cloud users are not the only ones struggling to keep costs down - providers are also doing so, for competitive reasons. For providers, it's critical to avoid the evil twins of under- and over-utilization. In the former case, machines sit idly, wasting energy and space, whereas in the latter, congested resources lead to violations of service-level agreements, incurring penalties and leave customers unsatisfied.

Market places are one of the most effective tools to control fluctuating demand and provide a win-win situation for consumers and service providers. In fact, almost all other major utilities, including electricity and water, are traded on market places. Therefore, it comes at no surprise that Amazon, among other providers, starts to offer its cloud computing resources on market places. Amazon calls its market for virtual machines "Amazon EC2 Spot instances". Instead of paying a fixed price, users bid the top price they're willing to pay for one hour of use of a virtual machine instance. As long as the current market price is lower than the bid, the user will be able to use the machine. For Amazon's Spot market place, we found that for the last 3 years, prices are on average 9× lower than Amazon's EC2 fixed prices (also referred to as on-demand market). However, the price differential can sometimes be as high as 30× more than the fixed prices.

Though the advantages of a market place for virtual machines are compelling, a fundamental difference exists between market prices for utilities such as electricity and those for virtual

[1] Brown University, Providence, RI, USA

machine instances such as EC2. If a utility price increases, the user can choose to use less electricity by turning off appliances. On Amazon's Spot Market, the machine automatically shuts down and becomes unavailable as soon as the price increases above the bid, resulting in a distinct form of machine failure. Whereas failures in the on-demand market (i.e., the fixed-price market) are unlikely, they are the norm with spot instances, and can be as extreme as having a failure every few hours or even minutes, depending on the bid price. Furthermore, machines within the same category (that is, the same type, OS, and region) and bid price usually shut down together because they observe the same price fluctuations, making failures not only highly correlated within a market, but also different from "normal" failures in the cloud.

In this paper we address the question, how analytical systems with different fault-tolerance strategies should be deployed on cloud market places. While recent studies looked at the "optimal" bid-price [An10] or how to adjust existing systems, like Hadoop or MySQL to spot instances [Ch10, Bi15], none of them systematically studied the implications of spot markets on analytics, specifically when using different fault-tolerant strategies. Instead, existing work is rather ad-hoc and heavily tailored towards Amazon's current market rules.

In this paper, we first systematically analyze and model Amazon's current spot market rules as well as discuss possible variations of it. Based on the results, we make theoretically sound recommendations for three common fault-tolerance strategies: (1) *fail-stop-redo* as used by traditional database systems, (2) *check-pointing* as implemented by Hadoop-like systems [Had, Bu10], and (3) *lineage-based* recovery as, for example, used by Spark [Za10]. Furthermore, we evaluate all strategies using real-world data sets obtained from Amazon's spot market place.

The remainder of this paper is organized as follows: In the next section, we summarize Amazon's current market rules and describe some of the unique characteristics we found. In Section 3 we abstract away from Amazon's rules and develop a more general market model, which we believe will also remain valid in the future. In Section 4 we use this general model to make recommendations on how to deploy systems with varies kinds of fault tolerance strategies. Within this section we also validate our results using real-world traces from Amazon Spot Market. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 Amazon's Current Spot Market

Currently, Amazon offers the biggest cloud market place. In this section, we first describe Amazon's market model in more detail, before we generalize it in the next section.

2.1 Overview of the Market Rules

Amazon offers the same machine configurations on the spot market as on their on-demand/fixed-price platform: general purpose (m), compute intensive (c), memory intensive, etc., each targeting different use cases. Each category contains different specific machine configurations varying in the main memory and number of virtual CPUs etc (e.g., one configuration called *c3.8xlarge* has 32 virtual CPUs, 60GB of RAM and two 320GB SSDs).

Most configurations are offered in eight global regions and can be used with any of the four supported operating systems.Users can freely pick the machine, region, and OS they want. In this paper, we refer to a specific combination of the machine configuration, OS and location as an *instance type*. Every single instance type has its own independent market and, thus, price. That is, when an instance type running on the East Coast with Linux is expensive, the same machine with Linux on the West Coast can still be cheap. In 2015, Amazon had a total of 1640 different markets (i.e., prices).

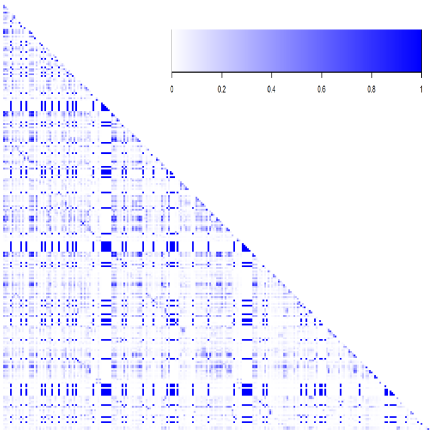


Fig. 1: US East1 Market (pearson) correlation heatmap with 1 min data interpolation of the latest 2 months. Every dot is the correlation between two machines on the spot-market. The darker the color, the stronger the correlation.

Users can buy machine time by bidding the maximum price they are willing to pay for an hour of use. Assume, that the user bids $0.75 per hour for an c3.8xlarge instance running Linux in the US-East region. If the user's bid price is higher than the selected spot instance's current price, then the instance is launched - (usually) in less than a minute. If the consumer's bid price is less or equal to the selected spot instance's current price, the request is pending. In this case, the user can either cancel the request or wait until the spot instance price drops below the bid. It is also possible to bid on N machines of the same type as a *bundle*, which ensures that the bid is either successful for all N machines or for none.

Machines are billed based on the Spot price at the start of each instance-hour [Am]. For example, if a $1 bid is successful at 2pm at a price of $0.7, which shortly after drops to $0.5, the user still pays $0.7 for the hour of use. Even if the user has a successful bid and is able to reserve the machine, there is no guarantee that it is usable for the entire hour. If the price increases during the hour above the bid price, the machine is shut-down immediately resulting in a distinct form of machine failure.

An important peculiarity of the current Amazon Spot Market is that, if Amazon shuts a machine down, the user does not need to pay for the partially used hour. For example, if a machine shuts down after 30 minutes due to a price increase, the user pays nothing; if it shuts down after 90 minutes, the user pays for only one hour. Yet, if the user himself shuts down the machine before the end of the hour (and prior to any price increase that would have caused a shut-down), the user always has to pay for the full hour regardless of the price increase.

2.2 Amazon Failure Characteristics

When the price of a specific instance increases above a successful bid of a user, the Spot instance is shut down, resulting in a distinct form of failure. In the following, we describe our finding of studying spot failures over a period of almost 2 years:

Failures are correlated: In contrast to traditional hardware failures, machines with the same bid for a specific instance type on the Amazon Spot market usually fail together. If the price increases above the bid price, all reserved machines that have the same bid price fail, if the price increases exactly to the bid price some reserved machines might fail. However, even beyond a single instance type, failures are correlated. Figure 1 shows the price correlation between different markets (i.e., different machine types, OS and data centers) over 2 month. As shown, the demand for different instance types can be highly correlated (e.g., the prices between the same category in different data centers). Considering correlated failures is thus one of the main challenges.

Failures are unavoidable: In the spot market, the available number of instance is dependent on the on-demand market, making the spot market a two customer class auction. The primary class users of the on-demand market pay a fixed price (also referred to as a reserved price), whereas the second class of users, the spot instance customers, can only use the left-over machines. As a result, there is no way for the second class, the spot market, users to avoid failures as in the worst case the first class can consume all resources.

Other Peculiarities: We also noted that some markets have bursts of high demand, probably explainable by power users. However, we also observed that the price fluctuations in some markets suddenly stopped, or that some markets have a weird almost binary (high price, low price) behavior. Whereas we assume that in the former case, Amazon might have added additional machines creating oversupply, we assume for the latter case, that the market is currently simply too small and, thus, creates the binary market behavior. We also observed on several occasions, that less-powerful machines are more expensive than more powerful machines; i.e., the market is not arbitrage free.

2.3 Discussion

Given these observations from the previous sub-section, we found that advanced market-based bid strategies are not very effective right now when used for the spot market. Some recent work [Be11] even suggested, that Amazon generates the price at random from within a tight price interval via a dynamic hidden reserve price instead of using a real demand-driven auction that follows the $(N + 1)^{st}$ model.

As a consequence of the findings in [Be11] and our own observations, we decided not to develop bidding-strategies based on standard models. Instead, we model the price changes as a random process and only use high-level aggregates (e.g., the average mean-time-to-failure for a given bid price in the last x days as the main statistic). It should be noted though, that even many advanced stock trading models still use the same oversimplifying assumption that price changes are a random process. In the future, and with the maturity of the spot market, we might need to revisit this decision.

3 Cloud Computing Market Models

Amazon picked one configuration in the design space of possible market models but also keeps it open to change its rules in the future. Moreover, other providers might decide for a different setup of their cloud markets [Mu12]. In the following, we therefore discuss alternatives to Amazon's current Spot market rules, argue which rules are likely to persist

or change, and based on those considerations develop a more general market model, which we then use throughout the remainder of the paper.

3.1 Instance Termination

While, at the moment Amazon can terminate an instance at any time, it is also easy to envision a market in which the machine is assigned to a user for a fixed amount of time, here referred to as a *reserved market*. In this case, a successful bid would allow the user to use the machine for at least X minutes (ignoring real hardware failures). On the one hand, this market would certainly provide benefits to the customer because it simplifies the assumptions. On the other hand, reserved markets reduce the revenue for the cloud provider. Moreover, as mentioned earlier, Amazon offers the over-capacity from the on-demand market (1^{st} customer class) on the spot market (2^{nd} customer class). It is reasonable to assume, that if more resources are needed for the on-demand market, they are taken away from the spot market by automatically shutting down machines. Thus, if the cloud provider would guarantee the resources for a fixed amount of time, he might not be able to sell the machine for a higher price on the on-demand market.

We therefore conclude, that Amazon's current product offerings are reasonable and likely to be used as blueprint by other providers. However, we can not exclude, that there might be other market models in the future. Studying them is beyond the scope of this paper.

3.2 Billing Policy

Amazon rents its virtual machines per hour. Obviously, an hour is just an arbitrary value. In the remainder, we assume that the customer is billed at the beginning of every ε-th minute of use:

Definition. *$\varepsilon \in \mathbb{N}$ minutes is the billing interval and $T_{startup} \in \mathbb{N}$ the time from the initial start of the machine until useful work can be done.*

Note, that the unit of ε is arbitrary. In this paper, we assume minutes to simplify the presentation. Moreover, we assume that the required startup time of a virtual machine $T_{startup}$ is (significantly) smaller than ε.

3.3 Auction and Pricing Principle

Amazon seems to use a $(N+1)^{st}$-based auction algorithm. However, it is likely, that they use a different price-finding scheme [Be11] or might even change the strategy in future. To generalize the findings of this paper, we do not rely on the knowledge of the strategy but only require to be able to observe the auction price over time. Furthermore, without loss of generality, we define that the price is redetermined at the beginning of every hour.

Definition. *$c_s^t \in \mathbb{R}$ is the Spot price at time t for instance type s. If the user successfully reserves an instance at time t_0 and uses it for m minutes, she has to pay*

$$C_s(m) = \sum_{i=0}^{\left\lfloor \frac{m}{\varepsilon} \right\rfloor} c_s^{t_0+i\varepsilon}$$

if the user bid b stays above the spot price for the duration of the reservation, i.e.,

$$\forall t \in [t_0, t_0 + m] : c_s^t \leq b$$

3.4 Notifications and Failures

We now address how to include the different failure cases into our model. Currently, Amazon shuts down the machine immediately when the price raises above the bid price. We already know from conversations with the Amazon Spot team, that they are considering to introduce a *warning* prior to the automatic shutdown to give the user time to save its intermediate result and allow for a clean shutdown.

Definition. *The time $\omega \in [0, \varepsilon]$ is a notification to the user before the automatic shutdown.*

While an interesting concept, the ω time also implies a minimum running time of the machine. In the extreme case, when $\omega = \varepsilon$ the auction changes to a reserved market auction as defined in Section 3.1 with all its benefits but also disadvantages for the cloud provider. In the remainder, we assume $\omega \ll \varepsilon$.

Furthermore, Amazon offers the functionality to start n machines together (i.e., either the bid for all of them is successful or none). We further observed, that reserved instances from the same user and with the same bid price either fail together (e.g., if the price increases above the bid), or they stay active together. We consider this as a valid assumption, which should also hold for other market models.

3.5 Cost for partially-used Intervals

Amazon currently does not charge for a partially used hours, if the machine was shutdown by Amazon because of a price increase. Similarly to the billing interval, this is an arbitrary convention and should be relaxed. We therefore introduce the notion of a discount γ for partially used hour.

Definition. *Let $\gamma \in [0, 1]$ be the percentage the user has to pay for a partially used hour, t_0 be the starting time of the instance and m the minutes since the start until the price raises above the bid b, then the total cost is:*

$$C_s(m) = \sum_{i=0}^{\lfloor \frac{m}{\varepsilon} \rfloor - 1} c_s^{t_0 + i\varepsilon} + \gamma \frac{(m \bmod \varepsilon)}{\varepsilon} c_s^{(t_0 + \lfloor \frac{m}{\varepsilon} \rfloor \varepsilon)}$$

$$\gamma \in [0, 1] \wedge c_s^{(t_0+m)} > b \wedge \forall i \in [t_0, t_0 + m[: c_s^i < b$$

Note, that $\sum_k^l$ with $k > l$ defines the empty sum and that our model allows a maximum discount of 100% (i.e., $\gamma = 0$). Finally, in the equation above and for the remainder of the paper, we do not consider real hardware failures as they are relatively rare.

4 Deploying Analytical Systems on Spot Markets

In the previous section, we developed a generalized cloud market model for virtual machines, which not only covers Amazon's current market rules but will also remain valid for some

potential changes (e.g., the discount). Given the model, we now discuss how analytical systems that use different fault-tolerant strategies are best deployed on a cloud market for virtual machines. In particular, we consider the following three fault-tolerance/recovery strategies that are most common in today's systems: (1) re-execution (typically used by traditional distributed database systems), (2) checkpointing (used by Hadoop and more recent frameworks), and (3) lineage-based recovery (e.g., Spark [Za10]).

4.1 Workload and Problem Statement

There are many different types of workloads and ways to model them. Here, we only consider the simplest form of an analytical task. We assume a single analytical task j, which requires a pre-determined amount of CPU cycles R_{CPU} per ε and memory/disk space R_{space}. A single task j could comprise an entire workflow or a query plan, composed of various smaller jobs. There exist a lot of work on determining the resource requirements of queries and analytical workflows[He11b, He11a]. Considering that sub-components of a task potentially have different resource requirements is beyond the scope of this paper.

If not mentioned otherwise, we assume that the analytical job is embarrassingly parallel and only dependent on the given CPU power (e.g., in the case of Amazon referred to as virtual CPUs).
Definition. *A task t takes $R = R_{CPU}/I_{CPU}$ minutes on an arbitrary instance that meets the requirements related to R_{space} and CPU "power" of I_{CPU} per billing interval ε.*

While this is an oversimplification as we assume no $T_{Startup}$ cost and perfect parallelism, we believe that this first simplified model helps already to gain interesting insights and is a first step in the right direction. It should be noted, that this approach is much in-line with any other development of a more theoretical model for any new problem. For example, even today most theoretical inventory models use oversimplified Poisson-based queuing theory or the famous Black–Scholes model to detect arbitrage never considered irrational traders and yet, is still heavily used worldwide. The reason for these simplifications is, that it creates mathematical more convenient models, which are human-understandable and function as the foundation for more complex but also more realistic models. This here presented model falls into the same category and we hope that the model will eventually be extended to include variants of Amdahl's law.

Given the assumptions before, our goal is to find the optimal deployment strategy:
Problem Statement 4.1. *Given the resource requirements R_{CPU} and I_{CPU} for a task j find the optimal instance type s and the optimal number of machines n to minimize the total expected cost $E(C)$ associated with executing the task.*

Thus, we consider the cost as most important factor with others, such as the real execution time, coming later. In the following, we will show how to achieve this goal for the different fault-tolerant strategies. Independent of the fault-tolerant strategy, a simple observation can already be made:
Proposition 1. *It is never beneficial to shut down an instance before the end of the billing interval ε.*

Even though it might sound obvious, it is an important observation (we omitted a formal proof). Terminating an instance before the end of the billing interval does never save money

because the instance is billed at the beginning of the interval. On the contrary, it can even increase the cost if a discount for partially used units exists ($\gamma < 1$) because the longer the machine runs after the finishing, the higher the chance that the discount will be applied to the hour. As we will see later, this observation plays an important role for all deployment strategies.

4.2 Strategy 1: Re-Execution

As a first fault-tolerant model we consider the simple *re-execution* strategy. Traditional database systems do not employ intra-query recovery — that is, if one or all machines fail during query execution, the analytical job/query must started from scratch. In the case of cloud markets, that even means that in case of failure the whole cluster with all its software has to re-start and re-execute the task.

The first observation we can make is:

Proposition 2. *A failure notification time ω has no impact on the deployment strategy*

This is a simple fact of the re-execution fault-tolerant model. A model which is not capable of re-starting, does not benefit from a shutdown notification.

However, an important question remains: What is the optimal deployment strategy. To tackle this question, we need to model the failures rates. We assume that the market price follows a Lévy process (e.g., a Wiener process) without a drift and an average price of $\bar{c_s^b}$ given a bid b. Note, that most financial market models are based on this assumption except that they might also consider a drift [Bl73]. For the remainder of this section, we assume a fixed bid price and simply write c for $\bar{c_s^b}$.

Furthermore, we model the price changes as a Poisson process. The bid-price induced failure event follows a Poisson process with intensity λ^b for the interval ε, a given bid price b and some instance type. It should be noted, that a Poisson process is a frequently used model for the arrival rate of failures. Different from other failure models (e.g., for hardware) where the likelihood increases over time, for bid price-induced failures there is no wear and tear making it a stochastic process with *independent* and *stationary* increments. Thus, the Poisson process is actually a better model for this type of failure than for "normal" failures.

Finally, it is relatively easy to determine the maximum likelihood estimator based on the history for both, λ^b and $\bar{x_s^b}$, by simply counting the time between failures and averaging the prices less or equal to b.

Giving these assumptions and assuming that $E(C_s^1)$ is the expected cost of running m machines of type s for exactly 1 billing interval ε to finish the job (we use capital C here to refer to the entire cost), the following holds:[2]

Proposition 3. *If $\lambda > 0$, $\gamma < 1$ and $(R_{CPU} \bmod q \cdot m \cdot I_{CPU}) = 0$, then running a task in a single billing interval ε is cheaper than running the job with fewer resources over several intervals until completion, i.e., in this case it holds that*

[2] Note, that we assume that we can perfectly split the task among the machines ($(R_{CPU} \bmod q \cdot n \cdot I_{CPU}) = 0$). For reasonably large tasks and with Amazon's large spectrum of machine types, this is reasonable. However, for smaller tasks the bin-packing problem becomes an issue and remains future work.

$$qE(C^1) \leq E(C^q)$$

The intuition behind this result is as follows: As long as a failure rate and a discount exist, the cost of a successful task is the same, but every failed task costs more as the user has to potentially pay the full price for machines even though he did not receive any usable result. More formally, let us assume that $q \cdot m$ is the number of machines to run the job in exactly one billing interval and m the number to run the job in exactly q intervals. In both cases, the total cost for a successful run is $C_{suc} = qmc_s$ and, thus, it is not important if the job is done in q or in 1 interval. However, the average cost for an unsuccessful run is:

$$\begin{aligned} &\sum_{k=1}^{\frac{\varepsilon R}{n}} \left(\underbrace{P(X^{t=k} > 0 \cap X^{t<k} = 0)}_{i} \left(\underbrace{cn \left\lfloor \frac{k}{\varepsilon} \right\rfloor}_{ii} + \underbrace{cn\gamma \left(\frac{k}{\varepsilon} - \left\lfloor \frac{k}{\varepsilon} \right\rfloor \right)}_{iii} \right) \right) \\ &= \sum_{k=1}^{\frac{\varepsilon R}{n}} \left(\underbrace{\left(1 - e^{-\frac{\lambda}{\varepsilon}}\right) e^{-\frac{\lambda(k-1)}{\varepsilon}}}_{i} \left(\underbrace{cn \left\lfloor \frac{k}{\varepsilon} \right\rfloor}_{ii} + \underbrace{cn\rho \left(\frac{k}{\varepsilon} - \left\lfloor \frac{k}{\varepsilon} \right\rfloor \right)}_{iii} \right) \right) \end{aligned} \tag{1}$$

Here $X^{t=k}$ is a discrete random variable describing the number of failures in minute k. To determine the average cost, we sum up the likelihoods that the instances fail in a particular minute during the execution (i) times the cost for a failure during that particular minute (1-(ii) and 1-(iii)). The last minute before the job finishes can be calculated as $\varepsilon R/n$ as we normed the CPU requirements to one billing interval. The failure likelihood $1-(i)$ is simply the probability mass function for at least one failure in a minute, $(1 - P(X = 0))$, times the likelihood that we had no failure in the previous $(k-1)$ minutes, $P(X = 0)^{k-1}$. Note, that we make use of the fact, that the Poisson distribution is *infinitely divisible*. That is, we can divide λ by the billing interval length ε to calculate the likelihood of a failure per minute. The cost of the failure consist of two parts: $1-(ii)$ defines the cost for every completed billing interval, whereas $1-(iii)$ describes the cost for the partially used hours. Note, that $(k \bmod \varepsilon) = k/\varepsilon - floor(k/\varepsilon)$. A transformation of equation 1 leads to:

$$cn\left(e^{\frac{\lambda}{\varepsilon}} - 1\right) \sum_{k=1}^{\frac{\varepsilon R}{n}} \left(\underbrace{e^{-\frac{\lambda(k-1)}{\varepsilon}} \left\lfloor \frac{k}{\varepsilon} \right\rfloor (1-\gamma)}_{i} + \underbrace{e^{-\frac{\lambda(k-1)}{\varepsilon}} \frac{k}{\varepsilon}}_{ii} \right) \tag{2}$$

Thus, if no discount is given ($\gamma = 1$) (i.e., the user has to pay 100% for partially used hours), the first part of the equation $2-(ii)$ equals zero and it does not matter if the task runs for more than one billing interval. However, if $\gamma < 1$ than allocating more machines to finish in one hour (or even less) is always beneficial as it guarantees that $\frac{\varepsilon R}{n} < \varepsilon$ and therefore, $\forall k \in [1, \varepsilon] : \lfloor \frac{k}{\varepsilon} \rfloor = 0$, forcing again to zero-out $2-(i)$.

Given that, we can make another even more interesting observation:

Proposition 4. *Using more machines to finish before the end of the billing interval ε can be cheaper than using fewer machines.*

At a first glance, this is counter-intuitive: Finishing earlier and wasting resources can be cheaper than using the resources optimally until the end of the billing interval. The reason lies yet again in the discount in the case of a failure. In order to prove the proposition, we first need to derive the total expected cost for executing a task j.

From proposition 3 we already know, that executing a task in a single billing interval is always beneficial. Thus, we can restrict ourselves to one billing interval only. Based on equation 2 we can derive the expected cost for a failure between minute a and b as:

$$\begin{aligned} EIC(a,b) &= nc \sum_{k=a}^{b} \frac{k}{\varepsilon}\gamma P(X^{t=k} > 0 \cap X^{t<k} = 0) \\ &= nc \sum_{k=a}^{b} \frac{k}{\varepsilon}\gamma \left(1 - e^{-\frac{\lambda}{\varepsilon}}\right) e^{-\frac{\lambda(k-1)}{\varepsilon}} \end{aligned} \tag{3}$$

As before $\left(1 - e^{-\frac{\lambda}{\varepsilon}}\right) e^{-\frac{\lambda(k-1)}{\varepsilon}}$ is the likelihood of a failure in minute k. Thus, every minute multiplied with the cost of n machines with cost c times relative discount $\gamma\frac{k}{\varepsilon}$ for minute k leads to the cost of the particular minute, and accordingly the sum over all the minutes in the time interval to the total cost of the interval. Given the expected cost for a failure in a specific time interval, the overall expected cost can be calculated as:

$$\begin{aligned} C = & \underbrace{\sum_{i=1}^{\infty} P\left(X^{t \le \frac{R}{n}} > 0\right)^i EIC\left(1, \frac{R}{n}\varepsilon\right)}_{(i)} + \\ & \underbrace{EIC\left(\frac{R}{n}\varepsilon + 1, \varepsilon\right)}_{(ii)} + \underbrace{ncP\left(X^{\frac{R}{n} < t \le \varepsilon} = 0\right)}_{(iii)} \\ = & \underbrace{\sum_{i=1}^{\infty} \left(\left(1 - e^{-\frac{\lambda R}{n}}\right)^i\right) EIC\left(1, \frac{R}{n}\varepsilon\right)}_{(i)} + \\ & \underbrace{EIC\left(\frac{R}{n}\varepsilon + 1, \varepsilon\right)}_{(ii)} + \underbrace{nce^{-\lambda\left(1-\frac{R}{n}\right)}}_{(iii)} \end{aligned} \tag{4}$$

Here (i) is the expected cost of all failed attempts, (ii) is the expected cost if the task completes but a failure occurs before the end of the billing interval, and finally (iii) is the full cost for the billing interval times the likelihood that no failure occurs before the end of the billing interval.

Maybe surprisingly, it is possible to derive a closed-form solution and the first derivative of equation 4, but it is lengthy and thus, we omit to show it here. Instead, we discuss a special case of the parameter space, partially used hours after failures are free (i.e, $\gamma = 0$), to demonstrate that using more machines than necessary to finish in one billing interval can be cheaper. Note, that this special case, is also Amazon's current model. After our

discussion, we again generalize the market model and explain why using more machines is also beneficial with any other discount (as long as a discount exists).

Theorem 5. *With a discount of 100% ($\gamma = 0$), the optimal number of machines for the re-execution strategy is:*

$$n_{\gamma=0} = max(\lambda R, R) \cdot m$$

In this formula, m are the number of machines require to finish R in one billing interval. Thus, m is a simple scale factor. To prove this theorem, we again consider equation 4 and for simplicity assume that R can be executed in a single time interval (e.g., $m = 1$). With $\gamma = 0$ the EIC from equation 3 is always zero and thus, the whole equation reduces to the cost that we have to pay for a successful task times the likelihood that we have to pay (i.e., the likelihood of no failure):

$$C_{\gamma=0} = nce^{-\lambda\left(1-\frac{R}{n}\right)} \tag{5}$$

Deriving the first derivative of $C_{\gamma=0}$ yields:

$$C'_{\gamma=0} = ce^{-\lambda\left(1-\frac{R}{n}\right)} - \frac{c\lambda Re^{-\lambda\left(1-\frac{R}{n}\right)}}{n} \tag{6}$$

Given that, the minimum is $n = \lambda R$. It should be noted, that we require the *maximum* from Theorem 5 as the equation does not model the effect of tasks running longer than one billing interval. This is also not necessary as Proposition 3 already showed, that it is never beneficial to run over more than one billing interval.

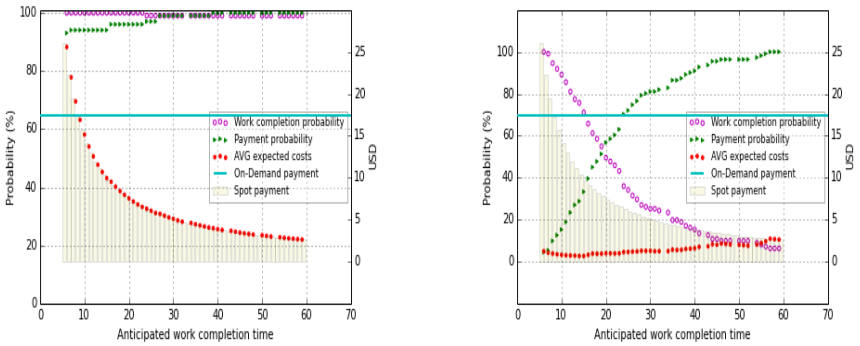
In the more general case of an arbitrary discount, the function in equation 4 remains convex and, depending on the failure rate and the discount, using more machines than minimally necessary to finish in a single billing interval can still be beneficial. In fact, the impact of parts (i) and (ii) of equation 4 on the optimum is very limited. That is due to the fact, that these cases are rather unlikely and thus, do not play an important role for the optimum.

4.2.1 The Optimization Goal

The last step of determining the optimum deployment for the re-execution strategy now depends on finding the optimal machine type. Again, we first consider the case with a discount of 100% ($\gamma = 0$). Using the optimum number of machines from Theorem 5 in cost equation 5 yields to the following optimization goal:

$$\underset{s}{\text{minimize}} \quad c \cdot min\left(\lambda Re^{1-\lambda}, R\right) \tag{7}$$

Looking closely at this minimization goal, and assuming a linear relation between the price of a machine and its failure rate (i.e., that the market values machines with a lower failure



(a) (b)

Fig. 2: Failure rate (λ) influence on work completion and payment probability, average expected cost, what we have to pay when we have to pay, on-demand cost, spot cost

rate), we can conclude that in an arbitrage-free market (i.e., a market where a more powerful machine is not cheaper), instance types with a higher failure rate will always reduce the cost and will potentially get the work done faster as we reserve more machines because of $n = \lambda R$. A surprising artifact created by the discounts for partially used hours. However, it should be noted, that our model does not consider the cost of starting the cluster and the software components (e.g., the services of the analytical system).

Similarly to the minimization goal of equation 7 , it is possible to define the optimization goal for market places with a discount of $0 < \gamma < 1$. However, due to space constraints we omit further details. Furthermore, we omit to consider $T_{startup}$ as part of the optimization because of space constraints (note, that $T_{startup}$ can be considered as a penalty function and easily be integrated).

4.2.2 Experimental Validation

While the goal of this paper are mainly the theoretical results, we also validated our findings using price traces from Amazon Spot Instance market. First, we analyzed the spot markets if there are actually really instance types which have on average more than one failure per hour (i.e., $\lambda > 1$). We calculated the statistics using a 5 day sliding window over 4 years of traces for every machine instance type (e.g., every data center, with every OS, etc) and different bid-prices. Not surprisingly, we found all types of failure rates (see also Section 2.2)

Here we show the results for the *us-east-1c–m1.large–Linux* instance type with on-demand price of \$0.175 and a bid price of \$0.0263 (15% of on-demand price) during two different time intervals $2012-09-12$ until $2012-09-17$ (Figure 2a), which had a lambda of $\lambda = 0.075$, and $2013-09-02$ until $2013-09-07$ (Figure 2b), which had a lambda of $\lambda = 1.8$. The former means, we had only 9 failures in total during the interval, whereas the latter refers to one failure every 30 minutes. We assumed a rather large task, which requires 101 instances to finish in 59 minutes (while 1057 machines are necessary to finish in 6 minutes).

We simulated the job execution using real Amazon spot traces from [Ja11] in every hour during the above mentioned interval. Figure 2a shows the that for the case with the low *out-of-bid* induced failure rate, using fewer machines is beneficial. That is, the work completion probability (the likelihood that the job finishes in the billing interval) stays almost constant independent on number of instances used. However, the probability that we have to pay (recall that Amazon has a 100% discount for failures), is also almost constant and only slightly drops with a lot of machines (e.g., to 95% with a 6 minuted anticipated completion time on 1057 machines). That is in almost all cases, the user would have to pay for the usage of the machine. As a result, using fewer machines to finish in exactly one billing interval (i.e., 60 minutes), has the lowest expected cost.

In contrast, when the failure rate is high the probability that the job completes quickly drops as fewer machines are used (see Figure 2b), while the likelihood that the user has to pay for a successful job increases. Looking at the expected cost reveals that there is a sweet spot at roughly 15 minutes execution time. However, according to our analyses the optimum should be at $\approx$ 30 minutes. The difference can be explained by the fact, that these are real world traces and that the failure likelihood heavily varies during the days. Furthermore, the difference in savings is only marginal between these two points. Finally, the figure shows that with such a high failure rate, it is actually a bad idea to use fewer machines: while the cost quickly decreases for the cases in which the user has to pay if no failures occurs (yellow bars in the background), the expected cost increases quickly as longer the computation takes.

4.3 Strategy 2: Checkpointing

The key, arguably counter-intuitive, insight the previous section made is that using more machines and thus, finishing a task earlier, can save money for analytical systems, that use re-execution to"overcome" failures, if the failure rate is high enough (more than 1 failure per billing interval). In this section, we study the optimal deployment strategy for systems that use check-pointing to reduce the failure recovery time, such as Hadoop. In Hadoop every output of a map-reduce job in a more complex in the query plan (i.e., data flow) is immediately check-pointed to HDFS. If a failure occurs, the checkpointed result is read from HDFS to continue. Our goal is not to model Hadoop's recovery strategy in detail, in contrary we want to abstract from it to keep the presentation simple and make our model also applicable to other systems. The surprising result of our analysis is, that with check-pointing the optimal deployment strategy completely changes: instead of exploring parallelism at all, it is the best possible strategy to use a single machine as long as possible.

First, it should be noted, that systems like Hadoop normally consider isolated failures of single machines. Though failures induced by markets normally cause multiple machines to fail (see Section 3) simultaneously.

In the following, we assume that a check-pointing system is able to recover all machines in t_R minutes from a checkpoint. Furthermore, we assume, that the whole system can checkpoint an output in t_C minutes and that t_C is smaller than the shut-down warning time: $t_C < \omega$. Finally, we assume that the number of machines can be re-adjusted after every (system-wide) failure.

Effectively that means, that except for the recovery time, it does not matter how often a machine fails. If we assume for the moment, that we can not effect the failure rate of a job to run on a specific machine, the question becomes again how many machines we should use. Again, for simplicity lets also assume, that bin-packing does not pose a challenge (i.e., the job can be divided into equal pieces). In this case, we can derive the following result:

Proposition 6. *The expected cost of using n or* $2 \cdot n$ *machines for a job is the same with check-pointing:* $\frac{R}{n} \geq \varepsilon$

The reason is simple: in contrast to re-execution, after a failure the machine can start where the previous instances left it of. This is not completely accurate since results are often cached and not directly checkpointed to disk. However, the additional time for re-execution is often minimal and we further on ignore it in our model. Therefore, it is no more the question of how much a failed attempt cost, instead the question becomes how often do we need restart the process to finish a job. If we reserve an instance on the sport market, on average we have to pay the following price per single instance:

$$\begin{aligned} E(C) &= \underbrace{\sum_{k=1}^{\varepsilon} P(X^{t=k} > 0 \cap X^{t<k} = 0)\frac{k}{\varepsilon}\gamma c}_{(i)} + \underbrace{P(X^{t<\varepsilon} = 0)c}_{(ii)} \\ &= \underbrace{\sum_{k=1}^{\varepsilon} \left(1 - e^{-\lambda}\right) e^{-\lambda(k-1)}\frac{k}{\varepsilon}\gamma c}_{(i)} + \underbrace{e^{-\lambda}c}_{(ii)} \end{aligned} \tag{8}$$

That is, we calculate the likelihood of a failure in minute k times the cost for a failure during that minute for the entire billing interval (i). In addition, we add the cost in the case there is no failure (ii). A small reformulation of equation 8 is able to remove the sum and yields to:

$$= \frac{c\delta e^{\lambda(-\varepsilon)}\left(-e^{\lambda} - e^{\lambda}\varepsilon + e^{\lambda+\lambda\varepsilon} + \varepsilon\right)}{\left(e^{\lambda} - 1\right)\varepsilon} + ce^{-\lambda} \tag{9}$$

Similarly, the expected number of minutes a single instance runs is given as (assuming it is not shut-down manually):

$$\begin{aligned} E(T)^{\infty} &= \sum_{k=1}^{\infty} kP(X^{t=k} > 0 \cap X^{t<k} = 0) \\ &= \sum_{k=1}^{\infty} k\left(1 - e^{-\lambda}\right) e^{-\lambda(k-1)} \\ &= \frac{e^{\lambda}}{e^{\lambda} - 1} \end{aligned} \tag{10}$$

In contrast, if the machine will be shutdown manually at the end of the billing interval ε, we get:

$$
\begin{aligned}
E(T)^{\varepsilon} &= \sum_{k=1}^{\varepsilon} k \left(1 - e^{-\lambda}\right) e^{-\lambda(k-1)} + e^{-\lambda}\varepsilon \\
&= \frac{\left(e^{\lambda} - 1\right) e^{-\lambda}\varepsilon + e^{\lambda(-\varepsilon)} \left(-e^{\lambda} - e^{\lambda}\varepsilon + e^{\lambda+\lambda\varepsilon} + \varepsilon\right)}{e^{\lambda} - 1}
\end{aligned} \tag{11}
$$

If the user plans to use n machines, we have to restart the machines (due to a failure) on average:

$$
\frac{\frac{R}{n}}{E(T)^{\infty}} = \frac{E(T)^{\infty} R}{n} \tag{12}
$$

and has thus to pay an average of:

$$
\left(\frac{\frac{R}{n}}{E(T)^{\varepsilon}}\right) nE(C) = E(T)^{\varepsilon} E(C) R \tag{13}
$$

That is we multiply the expected number of times we have to restart the machine, with the expected cost per machine times the number of concurrent machines n. As the average price in equation 13 does no longer depend on n our proposition holds. Note, that we use $E(T)^{\varepsilon}$ to account for the end of every billing interval. Also note, this multiplication is correct because of the special characteristics of the Poisson process: stationary and infinite divisible. Moreover, this equation only holds as long as $\frac{R}{n} \geq \varepsilon$.

Given these results we now know, that the expected cost of using one spot instance vs. using more instances to finish in a multiple of billing intervals is the same.However, there is still a significant difference between finishing in one or multiple billing-intervals: the risk the user has.

Theorem 7. *Using a single instance to finish a job in a single check-pointing interval is the cheapest and most risk-averse option.*

For example: lets assume the user decides to use 100 spot instances with a failure rate of $\lambda = 2$ and a cost of .1 to finish a job in exactly one billing interval (i.e., the machine might need to be restarted several times) and the discount is 100%. That is, usually — with his bid-price — he expects 2 failures per billing interval. According to our cost equation, this leads to an expected cost of $nE(C) = \$.1 \cdot e^{-2} = \$1.35335$. However, this is only the expected cost. Lets assume he got unlucky and the machines do not fail before the end of the billing interval. Now, he has suddenly to pay $n \cdot c = \$10$. A huge difference. In contrary, if he would have used a single instance for longer, the variance would have been much lower. This is do to the (strong) law of large numbers: If $\bar{C}_n = \frac{1}{n}(C_1 + C_2 + \ldots + C_n)$ and $E(C_1) = E(C_2) = \ldots = E(C_n)$ then for any positive number δ:

$$
\lim_{n \to \infty} P\left(\mid \bar{C}_n - E(C) \mid) > \delta\right) = 0 \tag{14}
$$

That is the likelihood to diverge from the expected mean goes to zero. In other words, as more intervals n we use, the more we have to pay for individual intervals, and the more stable is our expected cost.

Finally, similarly to the previous section, we need to determine the optimal machine. Here the optimization goal is not much different than before, except for the new cost function. Again on Amazon the machine with the best ratio of high failure rate and powerful hardware will lead to the maximum savings.

4.3.1 Experimental Validation

Again, we evaluated our findings using real traces from Amazon spot markets from the last 4 years. We simulated two types of deployment for every single instance type on Amazon Spot markets: (1) one deployment in which we allocated 100 instances to finish a job in 1 hour, and (2) another deployment in which we allocated 1 instance for 100 hours. We assumed that we would always use the median of the prices from the 4 years as the bid-price (note, that the results hold for any other bid-price). We then calculated the cost that the job would have caused the user based on the traces.

In Table 1 we show the results for three machine types, *m2.2xlarge*, *m2.4xlarge*, and *m2.xlarge* all from the *us-east-1a* data center. We omitted other instance types because they showed similar behavior, unless their failure rate was very low in which case there was no difference.

	1 instance for 100 hours		100 instance for 1 hours	
	$\bar{\mu}$ ($)	$\bar{\sigma}$	$\bar{\mu}$	$\bar{\sigma}$
m2.2xlarge	9	12	17	15
m2.4xlarge	15	18	32	30
m2.xlarge	5	5	11	7

Tab. 1: Simulated execution time and variance for two different deployments on two machine types

The result show what we expected. The cost standard deviation of using 1 instance is lower than of using 100 instances. However, we also noticed that the average cost is lower when running 1 instance for 100 hours than 100 instances for 1 hour. The reason is, that the failure rate does not follow a perfect Poisson process. Instead failures sometimes come in bursts at Amazon, while during other periods almost always the full price has to be paid. However, if we use 1 instance for 100 hours it better reflects the expected cost as we cover more billing intervals with a single job.

4.4 Strategy 3: Lineage

As a last fault-tolerance strategy we analyse *lineage*-based recovery, as for example used by Spark [Za10]. Lineage-based recovery keeps track of the lineage of every (partial) intermediate state. If a machine fails, the system recovers the missing intermediate state by re-executing only the work of the lost state. Lineage-based recovery has the huge benefit, that it has much less overhead than check-pointing and does not require to re-execute the compete query (but only parts of it). Unfortunately, lineage-based recovery has a severe draw-back if used on spot instances:

Proposition 8. *Lineage-based recovery does not protect from failures if used on n instances with a single spot instance type and bid-price.*

The reason is simple: as explained in Sections 2 and 3 a price increase of the spot price in this case either causes all machines to fail simultaneously or none. Thus, lineage-based recovery does not provide any benefit over re-execution for failures induced by the market.

Still, there exists one possibility to profit from lineage-based recovery on spot markets: We can combine different instance types, potentially with different bid-prices, into one portfolio, that is used to deploy the system and this way ensure, that not all machines fail at the same time. The challenge here is, that failures on today's spot markets are often correlated (see the discussion in Section 2 and also Figure 1).

Thus, this scenario requires a completely new optimization goal: We want to find the optimal portfolio, so that the expected cost is minimized

$$\underset{w_i}{\text{minimize}} \quad E(C) = \sum_i \omega_i E \tag{15}$$

with the cost variance of:

$$\sigma_p^2 = \sum_i \sum_j \omega_i \omega_j \sigma_i \sigma_j \rho_{ij} \tag{16}$$

where ρ_{ij} is the correlation coefficient between the cost on instance types i and j and $\rho_{ij} = 1$ for $i = j$.

We explored this option on Amazon's current market and our initial results are not very promising, that this is indeed a feasible option. Thus, we omit further details. The main reason is, that lineage-based recovery models often come with a significant overhead to even recover a single machine. For instance, Spark showed a 30% overhead in a cluster of 75 nodes. That is, to recover 60 seconds of work on a single machine, the **entire** cluster with all remaining nodes spend roughly 20s.

4.4.1 Experimental Validation

To better understand how machines are correlated in a portfolio, we selected 14 instances within a single region and with the same Unix OS. We then built all possible portfolios of sizes 2 to 14 using the 14 selected instance types. For each of the portfolios we again used the 4 years of Amazon traces to determine per portfolio size the portfolio with the minimum number of failures. Afterwards, for the "best" portfolio per portfolio size we calculated the number of times 1 instance type was failing alone, 2 instance types were failing together, 3 instance types were failing together, etc. using again the 4 years of Amazon traces. Figure 3 shows the result of this simulation using the real-world traces. As can be seen, building portfolios can help with reducing the correlated failures, but it is not a way to entirely overcome them. Furthermore, even though the correlation figure in Section 2.2 shows that for some machines no correlation exists, this is typically only true for instance types in different data centers or with different operating systems. Yet, it is questionable how useful it is to combine different instance types from different data centers or with different operating systems into one portfolio. As a consequence we believe that at least on the current Amazon Spot market building portfolios to take advantage of lineage-based recovery is not feasible.

	Number of failed instances at the same time (%)										
Size	1	2	3	4	5	6	7	8	9	10	11
1	100.0										
2	65.1	34.9									
3	48.5	40.8	10.7								
4	48.7	33.3	16.3	1.7							
5	39.1	36.0	19.0	5.8	0.1						
6	29.6	34.5	20.9	12.0	3.0						
7	25.0	34.9	22.3	11.8	4.9	1.1					
8	21.8	36.1	23.6	12.0	5.4	1.1					
9	23.8	33.5	20.3	11.3	6.1	3.9	1.1				
10	17.6	24.3	23.1	16.3	9.2	4.9	3.7	0.9			
11	15.7	21.6	21.2	18.5	10.6	5.2	4.6	2.0	0.6		
12	11.9	20.7	20.6	17.5	12.2	6.7	4.9	3.4	1.7	0.4	
14	7.6	17.7	19.9	17.6	14.3	8.8	5.6	4.1	2.8	1.3	0.3

Fig. 3: Portfolio: The percentage of one isolated instance type failures, 2 correlated failures etc, for the best portfolio of size 2 to 14 over the last 4 years. For example, for a portfolio size of 3, the portfolio with the lowest failure rate had 48.5% of single instance failures, in 33.3% of the cases two instance types failed together, and in 10.7% of the cases all three.

5 Related Work

Spot prices and bidding strategies: Various attempts have been made to analyse Amazon's spot prices and to create predictive models [Ja11, Be11, Ma11, Ta12a, Vo12]. For instance, [Be11] analyzed the cloud market and came to the conclusion, it might not be a real market, *yet*. However, this is pure speculation and was neither confirmed nor disputed by Amazon. In [Ja11] the authors extensively analyze the Amazon spot market. Furthermore, the authors model the price changes using a mixture Gaussian model to predict the cost of a given task. Different approaches for bidding strategies have been proposed and evaluated [Ta12b, Ta14]. All this work is orthogonal to our results. We developed neither a new bidding strategy nor a (yet another) predictive model. Instead, we tried to make general recommendations on how to use spot instances for different fault-tolerance models.

Checkpointing, Hadoop and spot instances: Similarly, there has been work on tuning the check-pointing interval for Spot instances [Yi12, Yi10, Kh13, Yi10, An10, Vo12, Bi15]. In this paper, we only considered the simplest form of check-pointing for which every single intermediate result is immediately written to disk. Note, that this is also the model Hadoop uses. Looking at other check-pointing strategies (e.g., check-pointing the intermediate state every 30 seconds) as done in [Yi12, Yi10, Kh13] is future work. There has also been work on deploying MapReduce, specifically Apache Hadoop, on Amazon spot instances [Li11]. However, their focus was on the required modification of the Hadoop architecture (e.g., to enable full cluster recovery) not on optimizing the deployment itself. Chohan et al. [Ch10] used a Markov chain model to predict the expected lifetime of the spot instance and have used spot instances only as potential task accelerators to improve the runtime of MapReduce jobs. Conductor [Al12] formulates cloud services as well as the performance of many data-parallel distributed computations as linear dependencies and uses dynamic linear programming for determining the optimal plan for deploying MapReduce jobs. Qin Zheng [Zh10] proposes provisioning new redundant copies for MapReduce tasks to improve MapReduce fault tolerance in the cloud (using Amazon EBS) while reducing latency. SpotAdapt [Ka15] automatically adapts to spot price changes and tries to find a better redeployment. Again, our goal was much more general since we theoretically analyzed the most cost-effective deployment strategy for different classes of systems.

Fault-tolerance Schemes: There is no shortage on systems and different types of fault-tolerance schemes. Traditional database systems, like MySQL or Postgres, typically do not use any inter-query fault tolerance and require to re-execute the whole query. Fine-granular fault-tolerance schemes are typically found in modern analytical systems, such as Hadoop [Had], Dryad [Is07]) as well as in many stream processing engines [Hw05, Ta06]. While stream processing engines check-point the internal state of each operator for recovering continuous queries, MapReduce-based systems [De08] such as Hadoop [Had] typically materialize the output of each operator to handle mid-query failures. For being able to recover, they rely on the fact that the intermediate results are persistent even when a node in the cluster fails, which requires expensive replication and prevents support for latency-sensitive queries. Other systems, like Impala [Cl] and Shark [Xi13] store their intermediates in main-memory in order to better support short running latency-sensitive queries. Spark uses the idea of resilient distributed datasets [Za12], which store their lineage in order to enable re-computation instead of replicating intermediates. We believe, that we were able to address a wide-range of these models in our analyzes and that the results can (easily) be adopted for any variation used by specific implementations.

6 Conclusion

A major aspect of future cloud-based computing platforms will be service and pricing models based on economic market principles. The Amazon Spot Market uses a bidding-based pricing model and is a pioneering step in this direction. If used wisely, spot instances can make analytics in the cloud significantly cheaper. In this paper, theoretically analyzed how analytical systems with different fault-tolerance strategies are best deployed on a cloud market place, like Amazon Spot, to reduce the overall cost. The key findings are: (1) that for systems, that do not implement any fault tolerance and have to re-execute the query, using more machines can be cheaper, whereas (2) for systems, that use an intermediate check-pointing strategy, such as Hadoop, a single machine is the best option. In contrast, (3) lineage-based recovery, as for example used by Spark, does not help on cloud market places. In addition to the theoretical results, we also backed-up the results using real data traces from the Amazon Spot Instance market.

References

[Al12] Alexander, W. et al.: Orchestrating the Deployment of Computations in the Cloud with Conductor. In: Networked systems design and implementaion(NSDI). 2012.

[Am] Amazon: , Amazon Spot Instances. `http://aws.amazon.com/ec2/purchasing-options/spot-instances/`.

[An10] Andrzejak, A. et al.: Decision Model for Cloud Computing under SLA Constraints. In: Proceedings of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). S. 257 – 266, 8 2010.

[Be11] Ben-Yehuda, O.A. et al.: Deconstructing amazon EC2 spot instance pricing. In: Proceedings 3rd IEEE Int'l Conf. on Cloud Computing Technology and Science. 2011.

[Bi15] Binnig, Carsten et al.: Spotgres - parallel data analytics on Spot Instances. In: ICDE Workshops. S. 14–21, 2015.

[Bl73] Black, Fischer et al.: The Pricing of Options and Corporate Liabilities. Journal of Political Economy, 81(3):637–54, May-June 1973.

[Bu10] Bu, Y. et al.: HaLoop: efficient iterative data processing on large clusters. In: Proceedings of the VLDB Endowment. 9 2010.

[Ch10] Chohan, N. et al.: See Spot run: Using spot instances for mapreduce workflows. In: USENIX HotCloud. 2010.

[Cl] Impala, http://www.cloudera.com.

[De08] Dean, Jeffrey et al.: MapReduce: simplified data processing on large clusters. Commun. ACM, 51(1):107–113, 2008.

[Had] Apache Hadoop. `http://hadoop.apache.org/`.

[He11a] Herodotou, H. et al.: , No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics, 2011.

[He11b] Herodotou, H. et al.: Starfish: A Self-tuning System for Big Data Analytics. In: In CIDR. S. 261 – 272, 2011.

[Hw05] Hwang, Jeong-Hyon et al.: High-Availability Algorithms for Distributed Stream Processing. In: ICDE. S. 779–790, 2005.

[Is07] Isard, Michael et al.: Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys. S. 59–72, 2007.

[Ja11] Javadi, B. et al.: Statistical Modeling of Spot Instance Prices in Public Cloud Environments. In: Utility and Cloud Computing (UCC) IEEE. S. 219 – 228, 12 2011.

[Ka15] Kaulakiene, Dalia et al.: SpotADAPT: Spot-Aware (re-)Deployment of Analytical Processing Tasks on Amazon EC2. In: DOLAP. S. 59–68, 2015.

[Kh13] Khatua, Sunirmal et al.: Application-Centric Resource Provisioning for Amazon EC2 Spot Instances. In: Euro-Par. S. 267–278, 2013.

[Li11] Liu, H.: Cutting MapReduce cost with spot market. In: USENIX Workshop on Hot Topics in Cloud Computing (HotCloud). 2011.

[Ma11] Mazzucco, M. et al.: Achieving Performance and Availability Guarantees with Spot Instances. In: HPCC. S. 296 – 303, 9 2011.

[Mu12] Murthy, M. K. M. et al.: Pricing models and pricing schemes of IaaS providers: a comparison study. In: Proceedings of the International Conference on Advances in Computing, Communications and Informatics. S. 143 – 147, 2012.

[Ta06] Tatbul, Nesime et al.: Load Management and High Availability in the Borealis Distributed Stream Processing Engine. In: GSN. S. 66–85, 2006.

[Ta12a] Tang, Sh. et al.: Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In: High Performance Computing and Communications (HPCC). S. 91 – 98, 6 2012.

[Ta12b] Tang, ShaoJie et al.: Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance. In: IEEE CLOUD. S. 91–98, 2012.

[Ta14] Tang, ShaoJie et al.: A Framework for Amazon EC2 Bidding Strategy under SLA Constraints. IEEE Trans. Parallel Distrib. Syst., 25(1):2–11, 2014.

[Vo12] Voorsluys, W. et al.: Reliable Provisioning of Spot Instances for Compute-intensive Applications. In: Advanced Information Networking and Applications (AINA). S. 542 – 549, 3 2012.

[Xi13] Xin, Reynold S. et al.: Shark: SQL and rich analytics at scale. In: SIGMOD Conference. S. 13–24, 2013.

[Yi10] Yi, Sangho et al.: Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In: IEEE CLOUD. S. 236–243, 2010.

[Yi12] Yi, Sangho et al.: Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. IEEE T. Services Computing, 5(4):512–524, 2012.

[Za10] Zaharia, M. et al.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 6 2010.

[Za12] Zaharia, Matei et al.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: NSDI. S. 15–28, 2012.

[Zh10] Zheng, Qin: Improving MapReduce fault tolerance in the cloud. In: Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW). S. 1 – 6, 4 2010.

Scientific Data and Hardware

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 383

Overview on Hardware Optimizations for Database Engines

Annett Ungethüm[1] Dirk Habich[1] Tomas Karnagel[1] Sebastian Haas[2] Eric Mier[1]
Gerhard Fettweis[2] Wolfgang Lehner[1]

Abstract: The key objective of database systems is to efficiently manage an always increasing amount of data. Thereby, a high query throughput and a low query latency are core requirements. To satisfy these requirements, database engines are highly adapted to the given hardware by using all features of modern processors. Apart from this software optimization, even tailor-made processing circuits running on FGPAs are built to run mostly stateless query plans with a high throughput. A similar approach, which was already investigated three decades ago, is to build customized hardware like a database processor. Tailor-made hardware allows to achieve performance numbers that cannot be reached with software running on general-purpose CPUs, while at the same time, addressing the dark silicon problem. The main disadvantage of custom hardware is the high development cost that comes with designing and verifying a new processor, as well as building respective drivers and the software stack. However, there is actually no need to build a fully-fledged processor from scratch. In this paper, we present our conducted as well as our ongoing research efforts in the direction of customizing hardware for databases. In detail, we illustrate the potential of instruction set extensions of processors as well as of optimizing memory access by offloading logic to the main memory controller.

1 Introduction

The relationship of hardware and software in general is at an inflection point. While software benefited from higher clock cycles of modern CPUs for many years, nowadays, hardware components are advancing at an incredible speed providing a rich bouquet of novel techniques. While this trend is opening up many opportunities for software solutions, they also pose significant challenges and risks. Specifically for database systems, we identify relevant advances in three different areas as well as their interplay as depicted in Figure 1: (1) *processing elements*, (2) *main memory*, and (3) *network*. On the area of *processing elements*, the core count increased and the internal techniques like advanced vector extensions, pre-fetching or branch-prediction improved within modern CPUs. Furthermore, alternative *processing element* approaches like GPUs, FPGAs, or any combination of them provide a wide field of opportunities to adopt database engine architectures and their implementation [He09, MTA09b]. On the area of *main memory*, the capacities increased over the last years allowing to keep the full (transactional) database in main memory [Fa12], while recent advances in the context of non-volatile RAM may be considered a disruptive technology for database systems with impact on data structure design, recovery mechanisms etc. [APD15]. On the *network* side, the network becomes complex due to an increasing number of cores. For example, in large NUMA (non-uniform memory access) multiprocessor systems, each

[1] Technische Universität Dresden, Database Systems Group, 01069 Dresden,
<fistname>.<lastname>@tu-dresden.de
[2] Technische Universität Dresden, Vodafone Chair Mobile Communications Group, 01069 Dresden,
<fistname>.<lastname>@tu-dresden.de

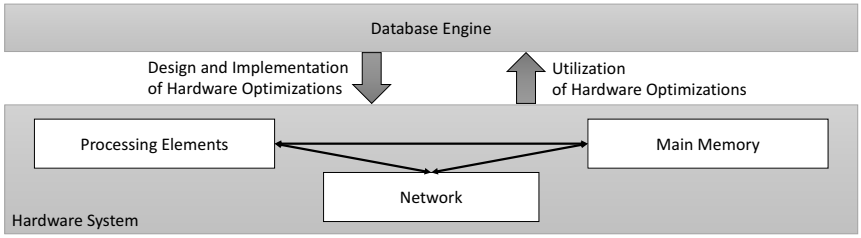


Fig. 1: Overview of Hardware/Software Co-Design for Database Engines.

multiprocessor has its own local main memory that is accessible by other multiprocessors via a communication network. This NUMA behavior may cause significant performance penalties when not considered correctly [KSL13], while Infiniband [Me07] or OmniPath [Bi15] provide bandwidth figures for communication between nodes close to performance characteristics of node-internal communication (e.g. via QPI [In08]). Considering higher latency in case of RDMA (remote direct memory access) [We15] for accessing memory of remote hosts, memory access of remote and local memory access (in case of multi-hop) are by now already exhibiting similar characteristics blurring node boundaries.

In general, all software components of database engines are highly specialized for the underlying hardware to satisfy the high performance requirements. Therefore, database engine developers usually need a deep understanding of all hardware features. Apart from this software optimization, hardware/software co-design approaches gain in importance, because tailor-made hardware allows to achieve performance numbers that cannot be reached with software running on general-purpose CPUs, while at the same time, addressing the dark silicon problem. Within this paper, we give an overview of our currently ongoing as well as already conducted research work to support database systems by customizing hardware components. Although we tackle this research field from different angles, we will focus on the perspective of processing elements, discuss optimization opportunities as well as consider the relationship between processing elements and main memory.

Our focus on customizing processing elements for data management solutions is mainly driven by two observations: (1) *thermal effect* and (2) *massive on-chip / on-socket parallelism*. First, the notion of „dark silicon“ [Es11] represents the consequence of thermal problems in traditional chip design. While silicon structures are getting denser, the complete die can no longer be supplied with sufficient power due to an increasing power density. As an effect, some chip area may end up unpowered („dark silicon“). These areas can be used for specialized circuits which can be activated on demand as an alternative to chip space housing generic functionality. While different application domains already exploit the dark silicon effect for special-purpose extensions (e.g. bio-inspired application, image processing or cryptographic support), the database community does not yet heavily consider this opportunity. Second, since data transfer very quickly becomes the bottleneck in data-driven applications, massive parallelism in combination with tight memory coupling techniques might be well suited for database systems to pre-process data sets before forwarding the data to more complex operators. However, thermal constraints again play a crucial role resulting in a need of extremely low-power processing elements.

The remainder of the paper is organized as follows: We begin with a short introduction of the Tomahawk architecture—reflecting the foundation of our work—in Section 2. Then, we outline some of our developed database-specific optimization for the processing elements in Section 3. In particular, we want to illustrate the potential of instruction set extensions of the core processors. Afterwards, we sketch preliminary research results in optimizing memory access by offloading typical memory access logic of data structures (e.g. filtering or search in trees) to the memory controller in Section 4. The objective of this offloading is to reduce the traffic on the chip's internal network and thus optimizing the latency of core database access primitives. Finally, we conclude the paper with related work and a summary in Section 5 and 6.

2 Tomahawk Architecture

The hardware foundation of our hardware/software co-design approach is the Tomahawk platform [Ar14c]. This platform is a heterogeneous multiprocessor system-on-a-chip (MPSoC) and has been developed at our university, whereby the primary focus was on supporting mobile communication applications. Nevertheless, the platform aims to be able to adapt for highly specialized tasks while being very energy efficient. Next, we briefly describe the overall MPSoC architecture, before we introduce more details about our enhancements.

2.1 Overview

Generally, the Tomahawk platform consists of two subsystems called control-plane and data-plane [Ar14c] as illustrated in Fig. 2. The control-plane subsystem comprises a CPU (traditional fat core), a global memory and peripherals, whereas this CPU is also called application core (App-Core). The App-Core is responsible for executing the application control-flow. The data-plane subsystem consists of a number of processing elements (PEs), each equipped with a local program and data memory. PEs are not able to access the global memory directly, instead a data locality approach is exploited using scratchpad local memory. That means, the PEs are explicitly isolated from the control-plane subsystem. The data-plane subsystem is used as an accelerator for the control-plane. Therefore, this subsystem can be seen as slave unit in the overall system architecture. Both subsystems are

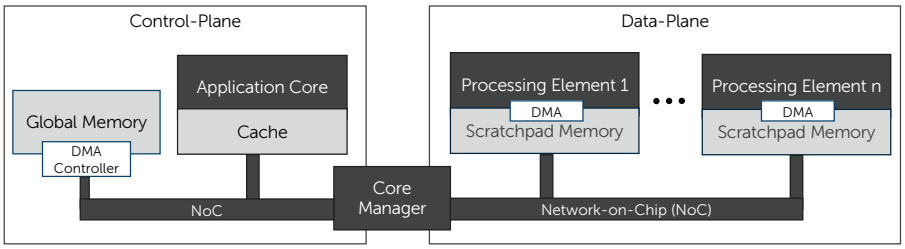


Fig. 2: An overview of the Tomahawk architecture.

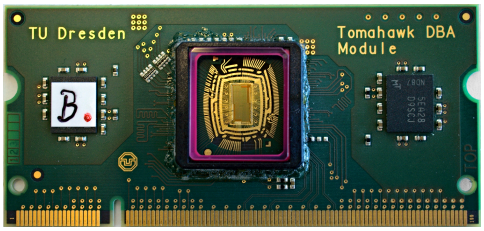


Fig. 3: The Tomahawk DBA module including the processor and two 64 MB SDRAM modules.

decoupled logically and linked together through a controller called Core Manager (CM). The CM is responsible for the task scheduling of the PEs, the PE allocation, and data transfers from global memory to the PEs and vice versa. Additionally, the CM can perform frequency scaling of the PE cores to minimize the power consumption.

Based on the overall collaborative setting at our university, we are able to tailor this Tomahawk platform for our database requirements. A number of modifications has been recently added to enhance data processing. This includes minor and uncritical changes like an enlarged scratchpad memory of the PEs as well as more sophisticated enhancements like specialized instruction set extensions for the PEs (see Section 3). To further increase the performance, we have also enhanced the DMA controller of the global memory to push down data intensive operations, like filtering or pointer chasing (see Section 4). The objective of this DMA enhancement (or intelligent DMA, iDMA) is to reduce the necessary data transfer between the control and data-plane which is clearly a bottleneck for processing large amounts of data. We have focused on the DMA controller of the global memory for several reasons. Nevertheless, the iDMA concept should be also beneficial for the PEs.

The overall development is conducted in cycles and the Tomahawk is currently in its fourth revision. The most recent available version is the third revision, also called the *Tomahawk Database Accelerator (DBA)* (Fig. 3) featuring 4 PEs. It is the first Tomahawk version that contains a specialized instruction set for database operations, which is further explained in Section 3. In the following, we describe the foundations for the processing elements and the DMA controller in more detail.

2.2 Processing Element Foundation

To not start from scratch, we use a Tensilica LX5 Reduced Instruction Set Computer (RISC) processor[3] for our processing elements offering a basic 32-bit ISA (instruction set architecture) with a base instruction set of 80 RISC instructions and one load-store unit. The LX5 is fully programmable with high-level programming languages, e.g. C/C++. Like every other synchronous microprocessor, it processes instructions in four steps: (1) Fetch Instruction, (2) Decode Instruction and Fetch Operands, (3) Execute, and (4) Store Results.

In step (1), the current instruction is *fetched* from the main memory into the *local instruction memory* and from the *local instruction memory* into the instruction *register*. In step (2), the

[3] Tensilica LX5 Datasheet - `http://ip.cadence.com/uploads/132/lx5-pdf`

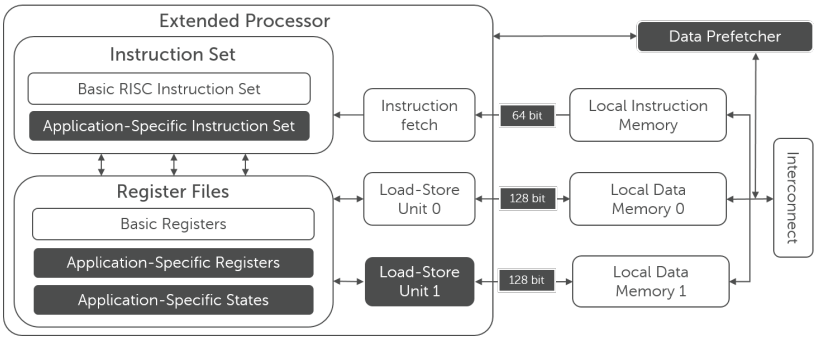


Fig. 4: The Tensilica LX5 Processor and the surrounding components have been extended by the features highlighted by a gray box.

instruction is decoded. This instruction belongs to the *instruction set* of the architecture. A result of the decoding is the determination of the necessary operands, which are then loaded from the *local data memory* into the corresponding *registers*. A *Load-Store Unit* is responsible for this step. In step (3), the actual computation is done. The results are stored in the corresponding *registers*. Step (4) finishes the instruction execution by storing the results in two phases. First, they are written back to the *local data memory*. Again, the *Load-Store Unit* is responsible for this phase. Then in the second phase, the result is copied from the *local data memory* into the main memory.

Besides the basic functionality, the Tensilica LX5 processor is configurable allowing us to modify and extend it according to the requirements of a database engine. A central extension which we added are an instruction set extension for database operators. Instruction set extensions are a common way for making frequent tasks faster, more energy efficient, and easier to use. Because of the different components which are involved in the four steps of the instruction execution, an instruction set extension does not only extend the machine code by plain instructions but also by architectural aspects like registers, data types, or memory modules if this is necessary. A well known example for instruction set extensions is SSE[4] for Single Instruction Multiple Data (SIMD) operations in x86 CPUs. In the second SSE revision, 144 new instruction were introduced together with enlarged registers from 64 bit to 128 bit, such that the values for two floating-point operations fit into one register enabling a faster double-precision processing[5].

For our database instruction set extension, we have added and modified a number of units of the Tensilica LX5 processor in general. An overview of the changed processor is shown in Fig. 4. In addition to the basic core, we have added application specific registers, states, and an application specific instruction set. These extensions vary for every developed extension, i.e. a bitmap compression extension and an extension for hashing have different instruction sets and registers. However, some modifications are shared by all instruction set extensions. In detail, the shared components are:

[4] Streaming SIMD Extensions
[5] Source: `http://www.intel.eu/content/www/eu/en/support/processors/000005779.html`

Time	Description	Latency in ns
t_{RCD}	Row activate to columns access delay	15
t_{RP}	Bank precharge to row activate delay	15
t_{CL}	Read column access to first data delay	15
t_{RAS}	Row activate to the next bank precharge delay	40

Tab. 1: Timing Constraints of the Micron DDR2 SDRAM (400 MHz)

Second Load-Store Unit We introduced a second load-store unit for being able to access two local memories simultaneously.

Data Prefetcher The data prefetcher is no part of the processor but operates next to it. It preloads data from an external memory into the local memories. This way it alleviates the memory access bottleneck.

Bus Width Extension The instruction bus is extended from 32 bit to 64 bit enabling us to introduce longer instruction words. The memory bus is extended from 32 bit to 128 bit for a faster memory access.

2.3 Main Memory Foundation

The global memory of our Tomahawk platform is of type DDR SDRAM (Double data rate synchronous dynamic random-access memory) - in detail, a Micron DDR2 SDRAM[6]. SDRAM devices use memory cells to store data, thereby the memory cells are arranged in two-dimensional arrays (matrices), also called banks. A typical SDRAM device has 4 or 8 internal banks. Therefore, bank index, row index and column index are necessary to locate a memory block in an SDRAM device. Furthermore, memory cells need to be periodically refreshed in order to keep the data due to the leakage currency of the capacity. This is known as refreshing, which can be simply done by a dummy read without output. According to the standard, every bit of an SDRAM device has to be refreshed every 64ms or less, which is periodically done by a refreshing logic.

Accessing an SDRAM device requires three SDRAM bus transactions besides the data transfers: *bank precharge*, *row activate* and *column access*. A *bank precharge* prepares the selected bank. A *row activate* command activates the word line selected by row index and copies the entire row in a row cache. Then one or more column accesses can select the specified column data using column index and output it through I/O gate logic. To access a different row of the same bank, the bank needs to be *precharged* again, followed by a new row activate and column access. Furthermore, SDRAM devices have a set of timing constraints that must be met for all transactions. For example, after a *bank precharge*, no transaction can be performed on this bank until t_{PR} (bank precharge to row activate delay time) has lapsed. Similarly, t_{RCD} is the minimal time interval between a row activate and a column access. For a read, the data will appear on the data bus t_{CL} (read column access to first data delay) cycles after the column access. Tab. 1 shows the timing constraints of our Micron DDR2 SDRAM.

[6] Micron DDR2 SDRAM - `https://www.micron.com/products/dram/ddr2-sdram`

SDRAM devices usually support burst mode allowing multiple sequential data within the same row to be accessed without additional column access transactions. The number sequential data in the burst mode is known as burst length. For DD2, the burst length should be set to 4 or 8. Because main memory accesses are cache misses, memory accesses always require entire cache lines. The burst length should be selected such that an entire cache line can be transferred in one burst. For an instance, for a system that has a 64-bit memory bus and 64B L2 cache line size, the burst length should be set to 8.

Generally, main memory requests issued by the CPU contain the address of requested memory block, the size of the block and type of the request (read or write). SDRAM devices can not process these requests directly. Thus SDRAM controllers, also known as memory controllers, are used to manage the flow of data going to and from the memory. In our case, we use a Synopsys DWC DDR2 memory controller[7]. Those controllers contain the logic necessary to read and write SDRAM devices. To locate a memory block in the SDRAM space, an SDRAM controller translates the address of requested memory block into the SDRAM address, which is composed of channel, rank, bank, row, and column index. With the considerations of timing constraints and possible bus contentions, the controller generates transactions to access the SDRAM device. For reads, the SDRAM controller returns the requested data to the CPU; for writes, it updates the requested memory block in the SDRAM device with the new data. Also, the SDRAM controller periodically refreshes the SDRAM devices to prevent data loss.

To summarize, SDRAM controllers and SDRAM devices compose our global main memory. SDRAM devices are organized to create an SDRAM address space. SDRAM controllers provide a memory interface through which the CPU or other devices (i.e. DMA controllers) can issue memory requests and receive responses. The DMA controller (direct memory access) is an additional feature that allows certain hardware subsystems e.g. processing elements to access main memory independently of the CPU making them an ideal candidate for customizing with appropriate logic. Without a DMA controller, the CPU is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work.

3 Processing Element Extensions for Database Primitives

To better support database engines from a hardware perspective, we developed certain instruction set extensions for database primitives like hashing [Ar14a], bitmap compression [Ha16c] and sorted set operations [Ar14b] over the years. In this section, we want to describe design questions, present instruction insides, and highlight an example instruction.

3.1 Design Questions

On the hardware level, the logic of an operation is implemented as a circuit consisting of one or more logic gates. For instance, an AND operation is realized with a single AND gate

[7] Memory Controller - `https://www.synopsys.com/dw/ipdir.php?ds=dwc_ddr2-lite_phy`

while a full adder uses two AND gates, two XOR gates, and one OR gate. It is also possible to merge several instructions into a new instruction. The input signals of such circuits are voltages which can be interpreted as *0* or *1*. If the input signal causes the circuit to close, there is a voltage at the output. This is interpreted as *1*. If the circuit is not closed, there is no voltage at the output. This is interpreted as *0*. The time it takes between switching on the input signal and getting an output signal is a hard constraint for the frequency of the circuit and therefore for the frequency of the whole core. The longer a circuit becomes, the longer it takes for getting an output signal. This results in a lower core frequency. Furthermore, we always have to consider the worst-case scenario, i.e. the longest path through a circuit. This scenario is called the *critical path*. Additionally, a complex circuit uses a larger chip area than a simple one. Thus, we have to find a compromise between the complexity of the circuit, i.e. the instruction, and the core frequency. This requires balancing different demands:

Universality: A common way for implementing a specialized instruction is to merge two or more basic instructions. If the resulting instruction is heavily used, this is a gain for the application because it can execute the instruction in one cycle instead of multiple cycles. But if this instruction merging is exaggerated to a point where the instruction is so specialized that it is rarely used, it wastes space on the processor and might decrease the clock frequency because of the potentially long *critical path*.

Technical feasibility: There are operations which are well suited for a hardware implementation while the corresponding software implementation needs many instructions, e.g. bit manipulations. For other operations, the tradeoff between a relatively slow software implementation and the increased required area and lowered clock frequency caused by a hardware implementation, has to be considered very carefully. For instance, an SIMD instruction for adding two vectors requires more space on the processor the larger the vectors get. Therefore the possible length of the vectors has to be limited.

For database operations this means that an instruction set extension has to cover the basic operators rather than a whole query. It might be possible to implement a whole TPCH query in hardware. It is very likely, that this quey would be processed much faster than the software implementation. However, this extension can not be used for anything else than the exact implemented query, while it occupies a significant amount of the area on the processor and has a longer *critical path* than a basic operator.

Furthermore, operators like a join or an intersection are by default no atomic operations. They can be split into several small steps. Each of these steps can be implemented separately resulting in a dedicated instruction for each of them. In the most extreme case, these instructions are a subset of the processor's basic instruction set and would therefore not result in a performance gain. Vice versa, packing all steps into one single instruction would clearly violate the demand for a certain universality. For this reason, we investigated the database operations, which we planned to implement. Additionally we identified and classified the working steps they have in common (Sec. 3.2). This way, we simplify the decision whether a step should get an instruction on its own or be merged with another step.

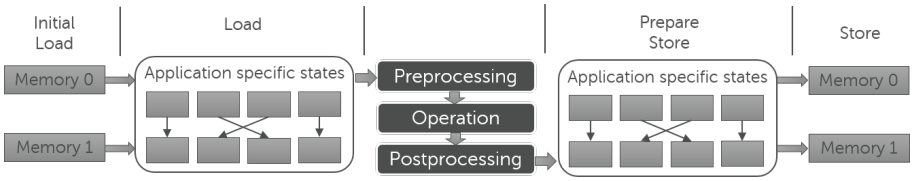


Fig. 5: Execution of a database operation using our instruction set extension

3.2 Instruction Insides

An instruction does not only consist of the logical operation itself, e.g. comparing two records, but also contains load and store operations and some pre- and post-processing. Therefore, our database operations contain the following steps (see also Fig. 5):

Initial Load Each load-store unit loads up to 128 bits from the memory into the intermediate states of the processor.
Load If necessary, the data is (partially) reordered and moved to a second internal state, e.g. for getting a 128-bit aligned line.
Pre-processing During the pre-processing phase all values for the actual operation are computed, e.g. the address of a bucket or the branch which will be chosen.
Operation The actual computation is done.
Post-processing Information for storing the data is collected, e.g. if a previous result will be overwritten.
Prepare Store If necessary, the results are shuffled. Afterwards they are stored into the dedicated states of the processor.
Store The results are written back to the local memory.

Following the demand for universality and a short *critical path*, the load operations are decoupled from the actual operation and store phase. Instead they are implemented as separate instructions. This way they can be reused on operations which request the same input data types. Since we have two memory modules and load-store-units, we can finish two load instructions within one cycle.

The operation and store phases are either separated or combined into a single instruction depending on the database primitive. Because of the complexity of some database primitives, the corresponding instructions are often scheduled over multiple cycles for obtaining the best compromise between execution time and latency. The number of the necessary cycles is determined empirically.

3.3 Example Extension

We realized several extensions implementing the introduced steps for different database primitives and added these extensions to the PEs of the Tomahawk. The natural decision was to start with some basic operators every database system needs to implement. For this purpose, a Merge-Sort has been developed as well as some operators which work

	Bitmap Compression and Processing (AND, OR, XOR)			Hashing						Sorted Set Operations				
Extension / Processor	WAH	PLWAH	COMPAX	Hash + Lookup	Hash + Insert	Hash Keys	Hash Sampling	CityHash32	Merge Sort	Intersection	Union	Difference	Sort-Merge Join	Sort-Merge Aggregation (SUM)
BitiX	X	X	X											
HASHI				X	X	X	X	X						
Titan3D						X			X	X	X	X	X	X
Tomahawk DBA	X					X	X		X	X		X		

Fig. 6: Overview of our extensions and the processors they are implemented on [Ar14b, Ar14a, Ha16c]. The processors which have already been manufactured at this point are highlighted.

on the produced sorted sets, i.e. Intersection, Difference, Union and Join [Ar14b]. Two processor designs featuring instruction set extensions for hashing [Ar14a] and compressed bitmap processing followed [Ha16c]. The first manufactured processor containing a subset of these extensions is the *Titan3D*, which was successfully tested [Ha16b]. Finally, the *Tomahawk DBA* was built with four extended cores [Ha16a]. A summary of the developed extensions can be found in Fig. 6.

One of the extensions which are available on real hardware is the processing of compressed bitmaps. Bitmaps can be used for a fast evaluation of query results. This is especially useful when the number of distinct values is small. Then, a match between the rows and an attribute can be stored as bitmap with a set bit indicating a match. An operation then only has to work on the bits and not the attributes themselves. We developed extensions for the compression and decompression of bitmaps using the *Word-Aligned Hybrid (WAH)* code and for basic operations like *AND* and *OR* on the compressed bitmaps [WOS04]. Furthermore, we implemented the same operations for two additional bitmap encodings, namely PLWAH [DP10] and Compax [FSV10].

The WAH code is used to encode bitmaps with run length encoding (RLE). It does not use signal words but encodes the necessary identifiers within every word [WOS04]. A WAH compressed bitmap can contain an uncompressed word (literal) or RLE compressed words (fills). The distinction between literal and fill is done by reading the first bit. A 0 indicates that this word contains a literal, a 1 indicates that it contains a fill. If it contains a literal, the remainder of the word represents the uncompressed word. If the word contains a fill, the second bit shows if set or unset bits are compressed. The remainder of the word is used to store the number of compressed chunks. For instance, the compressed word 11000010 expands to 1111111 1111111. The first bit indicates that this word is a fill. The second bit shows that the fills are set bits. The rest of the word shows that there are two chunks encoded. In the original work and in our implementation, a word is 32 bits long and a chunk contains 31 bits.

While operations on uncompressed bitmaps are memory bound, operations on compressed bitmaps are more complex, while having less data, therefore, these operations tend to be computation bound. Both cases have been tackled by our solution following the steps of the previous section:

Initial Load Since a second load-store unit has been introduced and the bus width has been extended to 128 bits, four 32 bit words of two input bitmaps can be loaded simultaneously from the local memory into the intermediate states of the processor.
Load For ensuring a continuous availability, the data is reordered because the memory access is aligned to 128 bit lines.
Pre-processing The first bit of the first input word of every input bitmap shows if it is a fill or a literal. If it is a literal, the word is preserved. If it is a fill, the input words are overwritten with the corresponding set or unset bits.
Operation For literals a bit-wise operation is performed. Fills only need to compare one bit. We implemented the operations AND, OR and XOR. This is done 4 times since there are 4 code words in a 128 bit line.
Post-processing The result of the operation is written to the output bitmap. If the result is a literal, it is appended to the output. If it is a fill, the previous word is overwritten to represent one more chunk.
Prepare Store To use the 128 bit memory bus optimally, four 32 bit words are buffered in a dedicated register before writing them back to the local memory.
Store The four buffered words are written to the local memory.

Because there are two load-store units, two load instructions can be executed simultaneously. They finish within one cycle. The remaining steps are scheduled over 4 cycles. For this purpose, an internal pipeline has been added. The main loop of the operation takes the majority of these four cycles. This is why the operation and the store process have been merged into one instruction. Hence, a code doing an AND comparison between two bitmaps is depicted in Fig. 7.

resetInst();	*Initialize states*
WUR_opID(0);	*Write into user register defining the operation,* *0 indicates an AND operation*
...	*Setting pointers and auxiliary variables*
do{	
ldXstream(); ldYstream();	*Load next 128 bits of bitmaps,* *Together both load operations need 1 cycle*
WAHinst(); WAHinst(); WAHinst();	*Do bitwise operation 4 times since there are four* *32 bit words in the state,* *Each operation uses 1 cycle*
}**while**(WAHinst());	*Returns 0 if bitlist has reached the end,* *Check for condition uses one more cycle*
...	*Any left over words in state? If yes, process them.*

Fig. 7: Code snippet which calls and AND operation on two WAH compressed bitmaps

resetInst();	*initialize states*
doWAHinst(0, 1024);	*perform AND operation*

Fig. 8: Reduced code snippet using our library.

The cycle used for checking the condition at the end of the while loop can be avoided if the number of loops is precalculated and the while loop is replaced by a for loop. For simplifying the usage of the extensions, there is also a C-library hiding these instructions behind handy functions. Using this library, the example reduces to the code snippet as shown in Fig. 8.

The first parameter of `doWAHinst` in Fig. 8 is an `unsigned int` defining the operation and the second parameter is an `unsigned int` containing the size of a bitmap in bits. The pointers to the bitmaps are globally defined.

3.4 Experiments

To demonstrate the gains of our hardware implementation, we compare it to the Tensilica LX5 core which we used as the base for our PEs and to an Intel i7-6500U, which is a current low-power CPU. For the sake of comparability, we measure the single thread performance on the i7 since the LX5 has only one core. All test cases are written in C and compiled with the -O2 flag. The ISA extensions, which have not been manufactured, run in a cycle accurate simulator provided by Tensilica.

Compression: Fig. 9 shows the execution times of operations on differently compressed bitmaps. Additionally the compression and decompression times for the WAH compression are pictured. The values are averaged over different bit densities ranging from $1/2^0$ to $1/2^{16}$. Each input set has a size of 2500 elements. For every test case, the comparison shows that there is a speedup of two orders of magnitude if the extensions are used compared to not using them. The extensions even outperform the i7 significantly. The latter is even more surprising considering that the i7 draws between 7.5 and 25 W while the LX5 does not draw more than 0.15 W in any of our test cases. The exact performance gain varies

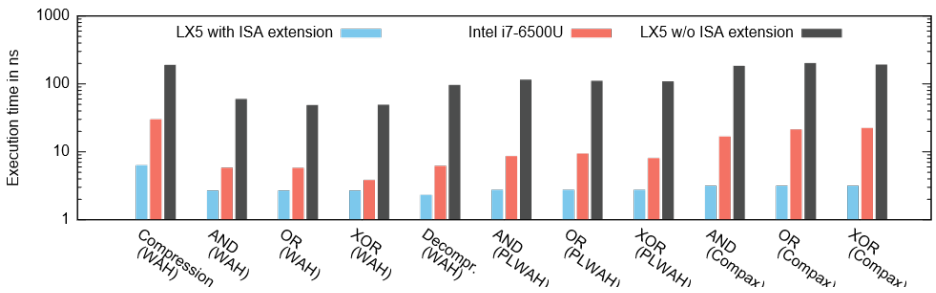


Fig. 9: Execution times for different bitmap compression techniques and operations on the compressed bitmaps

among the compression methods. For instance, the operations on WAH-compressed bitmaps are only twice as fast using our extension compared to the i7. However, using the more compute-intensive Compax algorithms, we can see speedups of 500% and more.

Further Primitives: Fig. 10 shows the throughput of some other selected operators, i.e. a merge-sort, some sorted set operations, and hashing. The input set size we used for the sorting and the sorted set operations is 5000 32-bit elements. That means, a set still fits into the local memory of a PE. The selectivity is set to 50%. In our experiments, we found the merge-sort to be performing worst compared to the i7. There, our extension does not reach the throughput of the i7 but stays within the same order of magnitude, while drawing only a fraction of the power of the i7. The merge-sort is the operation with the most accesses to the local memory since intermediate results have to be stored. These accesses have been accelerated by introducing the second load-store unit and increasing the data bus width, however, the possibilities for further optimization in this direction are more restricted than for the algorithm implementation itself. On the other hand, for operations on the resulting sorted sets, the memory accesses can be reduced and serialized. Hence, they outperform the i7 while still only draining 0.15 W at the maximum. The highest performance gain is reached with the hashing operators. There is a 100x speedup compared to the single thread performance on the i7 and even a 1000x speedup compared to the C-implementation on the LX5 without the ISA extensions. Hashing is also the application with the highest optimization potential starting as the lowest performing operation and becoming the most efficient operation when the respective extensions are used.

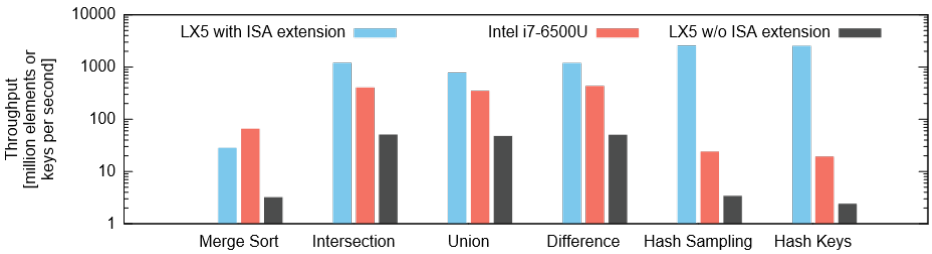


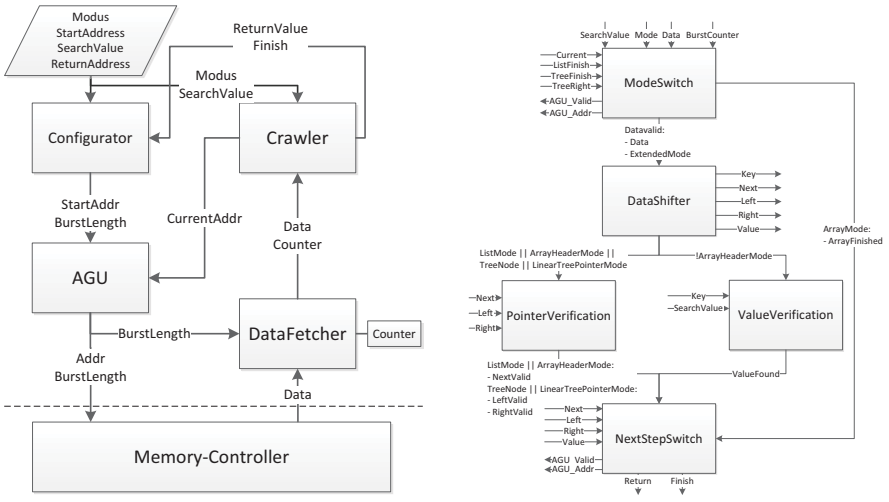
Fig. 10: Throughput for different operations.

4 Intelligent Main Memory Controller

In addition to the hardware optimization by means of special instruction set extensions for the PEs, access to the data in the global main memory can also be made more efficient. Instead of transferring massive amounts of data to the PEs for processing, e.g. filtering out data or searching a specific object in a data structure, we want to push-down the appropriate logic to the DMA controller. In this case, our intelligent DMA controller (iDMA) will be responsible to conduct fundamental data-intensive tasks like filtering or searching and the final result is only transferred to the PE.

In detail, we designed a first version of our iDMA approach which is able to answer point queries on various dynamic data structures. The simplest dynamic data structure is the linked list. With the help of pointers, appropriate objects are coupled one after the other. To extract a particular object (point query), the list must be searched from beginning to end using the

pointers. This evaluation is usually done on a PE which requires the transfer of all necessary memory blocks. By dynamically inserting and removing objects, the objects usually do not necessarily co-locate in main memory, which increases the number of transferred memory blocks from the global memory to a PE. Instead of extracting the object at the PE, our iDMA takes over this task and transfers only the desired object to the PE which reduces the data transfer massively. In addition to the linked list, our iDMA also supports hash tables with overflow chains, linked arrays, and the binary tree. For all of these dynamic data structures, our iDMA offers an enhanced point query interface.



(a) iDMA block diagram (b) Submodules of the *Crawler*

Fig. 11: Enhanced DMA controller.

To support point queries in our iDMA, the corresponding search logic for the different data structures is translated into hardware. Fig. 11(a) shows the block diagram of our enhanced DMA controller. The *Configurator* accepts requests and analyzes them to make a decision which actions must be performed. Thereby, different requests at the same time are possible. For the processing of point queries on dynamic data structures, the *Configurator* needs the following input: (i) the data structure to be processed, (ii) the address of the first object of the data structure, (iii) the search object (point query), and (iv) an address to which the result of the search is written. In the following steps, the enhanced DMA controller answers the point query on the corresponding data structure by scanning over the data as done in the software. In order to do that, additional components are required to load the necessary data by directly addressing the memory. One component is the *Address Generation Unit (AGU)*, which generates the necessary memory instructions and passes them on to the memory controller. If the delays of the memory controller and the memory are greater than the burst length, further burst requests must be pre-loaded for the some dynamic data structures. The *AGU* receives the calculated prefetch distance from the *Configurator* so that the preload can be executed.

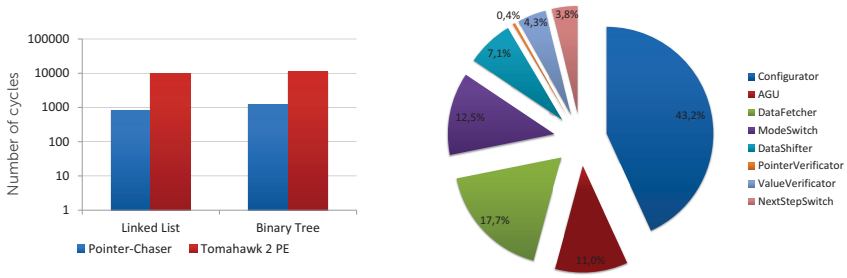
Then, the *DataFetcher* receives the data from the memory and counts the number of burst request replies. In addition, the *DataFetcher* compares the storage tags with the stored tags to assign the data obtained to the requests as multiple requests are possible. Then, the data with the assigned request is transferred to the subsequent *Crawler* module. This *Crawler* is responsible for complete data processing. For this, the crawler is divided into several submodules as illustrated in Fig. 11(b). On software or application level, an object is usually not processed until it is fully loaded. However, since the bandwidth of the memory is limited, an object can usually not be loaded in one clock cycle. But, there is the possibility to partially process the objects by prior knowledge of the dynamic data structure as done by our iDMA.

The *ModeSwitch* as depicted in Fig. 11(b) selects per clock the mode for the appropriate data structure and outputs it to the other submodules of the *Crawler*. Additionally, the *ModeSwitch* sends burst requests to the *AGU* for large data structures (e.g., linked arrays), since often a burst is not sufficient to load the entire object. For that, the *ModeSwitch* receives the necessary number of burst requests from the *Configurator* and the current burst position of the *DataFetcher*. The burst requests are scheduled in an appropriate way, so that the queue does not overflow in memory. Furthermore, the scheduling considers the time constraints as introduced in Section 2.3. Then, the *DataShifter* receives the data of the memory which was obtained by the *DataFetcher* and divides it into blocks of the size of 32 bits. We currently support only 32bit objects. Afterwards, the *ValueVerificaton* and the *PointerVerification* get the data. Here, the object is compared with the point query (search criterium) and the pointers are checked for validity. All signals are then sent to the *NextStepSwitch*, where they are evaluated before the next step is initiated. For example, the *NextStepSwitch* starts the transfer of a new object of the data structure or sets the finish signals so that the Configurator can terminate the search.

To summarize, our iDMA controller answers point queries on dynamic data structures (linked list, hash tables with overflow chains, linked arrays, and the binary tree) using a hardware-implemented pointer-chasing approach. To evaluate our approach, we integrated the iDMA in the Tensilica cycle accurate simulator, so that we are able to compare the iDMA approach with a software-approach running on a PE. Fig. 12(a) shows the results for a linked list and a binary tree, both are holding 840 32-bit objects. The SDRAM properties and timing constraints are described in Section 2.3. As we can see, our iDMA (pointer-chaser) outperforms the point query processing on the PE by a factor of 10 (in terms of number of cycles).

Fig. 12(b) depicts the percentage areas of the partial modules of our iDMA based on the developed hardware model. The *Configurator* takes about 43% of the area because the parameters of the search (search value and return address), the data structures, as well as other signals must be stored in registers. In addition, arithmetic units are added for the calculation of the required number of bursts. The *AGU* requires approximately 11% of the area, since the *AGU* needs registers to store the current memory tag which must be sent with each memory request so that the responses can be assigned. Since the *DataFetcher* has to assign the replies of the memory to the queries by matching their tags, the *DataFetcher* needs a FIFO for buffering all requests from the *AGU* resulting in 11% of the area. Like

the *AGU*, the *ModeSwitch* of the *Crawler* requires several registers and arithmetic units for processing. This results in the required area of approximately 12.5%. The remaining four submodules of the *Crawler* only process signals and make decisions so that no data is stored or arithmetic is executed. Therefore, their surface areas are not so large.



(a) Execution times in number of clocks of iDMA with pointer chaser and a processor with an SDRAM

(b) Percentage area fraction of the submodules of the pointer chaser component

Fig. 12: Evaluation of Pointer Chaser Component.

5 Related Work

Fundamentally, hardware/software co-design can be done for all hardware components of (a) the processing elements, (b) the memory, and (c) the network. In this section, we want to discuss different approaches:

Processing Elements A common technique for the extension of processing elements—which is widely used in commercial hardware systems—is the extension of the basic instruction set of standard CPUs with use-case specific frequently used instructions. Apart from the widely known SSE extensions, Intel® also developed extensions for other use-cases, e.g. the Software Guard Extensions (SGX) [An13] or the Advanced Vector Extensions (AVX) [Fi08]. While AVX is basically an enhancement of SSE4, SGX serves a completely different domain. It introduces protected software containers, called enclaves, for securing sensitive content. The access to these enclaves is protected by hardware control mechanisms. Thus, not performance but security is the optimization goal for the SGX extension. This is especially useful in distributed applications and client-provider-scenarios, and holds the potential for paradigm changes on the networking layer (c). There are also extensions developed by ARM®, e.g. THUMB which is now a de-facto standard extension for embedded ARM® cores or NEON which is an extension for SIMD processing.

A way for bringing very specialized functionality into hardware is using FPGAs, which contain a programmable logic circuit instead of a hard-wired one. Such FPGAs can serve as processing elements for very specialized jobs. Several works have used FPGAs for database applications [WTA13, MTA09a]. The challenges in programming an FPGA and developing an instruction set extension1 are similar in many aspects. For instance, a decision has to be made which functionality is implemented in software and which is applied to the FPGA, e.g. for boolean expressions, Woods et al. compute a truth table in software and send it to

the FPGA, where it serves as a lookup table for the query evaluation [WTA13]. However, the setup of a system using an FPGA is different from ours. While we enhance the hardware components themselves, i.e. the processors and the memory, an FPGA is an additional component which has to be connected to the host system. This introduces a data transfer overhead which has to be regarded when optimizing the query execution. Additionally a hard-wired circuit is usually much smaller than the programmed equivalent. This comes at the cost of reconfigurability. An implementation for an FPGA can be changed later whereas an instruction set extension is fixed once it is built. When it comes to the costs, an FPGA is expensive once when it is purchased while changing the configuration comes at almost zero cost. A processor or memory chip with additional hard-wired functionality becomes more affordable the more pieces are manufactured. This is because the initial modification of the manufacturing plant is the most expensive task, e.g. manufacturing the wafer. Hence, an FPGA is better suited for very specialized tasks which do not require a large quantity of chips whereas our approach is suitable for the mass.

Memory The bandwidth of memory buses and the capacity of (intermediate) memory is limited. In the memory hierarchy, bandwidth is growing more the closer the memory is to the processing element. But at the same time the capacity decreases. In data intensive applications this leads to high latencies which make the memory access a bottleneck in query processing. There are several approaches to overcome these architectural limitations.

One trend is to reduce the data that has to be sent to the processing elements. This is done by moving the functionality, which filters the data, closer to the memory. An early proposal using this concept is the *Intelligent Disk* (IDISK)[Ke98]. It combines a processor, DRAM and a fast network, on one chip (IRAM) which is put on top of the disk. This enables simple operations on the data before it is sent to the actual database engine. This way the data can be filtered before sending it to a smaller intermediate memory close to the main processing elements. Additionally the computing load for doing this filtering is taken away from the main processing elements. Instead of a relatively slow multi purpose processor, an FPGA can also be used as the component between the memory and the database engine. This has been implemented in IBM® *Netezza* [Fr11]. Here, an FPGA is installed between the memory and the processing nodes. It is responsible for filtering and transformation before the data is sent to the node. Finally, the already mentioned work by Woods et al. introduces a system called *Ibex* [WTA13], which uses an FPGA not as a usual processing element but as a filter between an SSD and the database engine.

Another trend is to decouple the dataflow from the actual processing in the processing elements. An approach for this direction is the SGI *global reference unit* (GRU). In the SGI Altix UV series, the GRU is a part of the UV Hub which connects the processors, the local SDRAM and the network interface. A UV Hub contains 2 GRUs which connect two processor sockets with the NUMAlink5 interconnect [SG10]. This way the whole system can be used as a single logical system including the memory modules. Since the GRU is programmable, it can be set up to execute instructions, e.g. remote copy of data, while the actual user application is doing something else.

However, the mentioned approaches focus on the interconnect between different sockets and their attached RAM or the database engine and the permanent memory, not on the connection between the RAM and the according processing element.

All Layers One step further than focusing on improving selected elements of existing systems, is to develop a completely revised architecture which can include changes on all 3 layers of the system design. On the large scale this includes heterogeneous architectures and changes in the system layout, e.g. the transfer of the memory controller from the north bridge to the CPU. On the small scale it addresses features like the out-of-order processing implementation. For instance, Hyperion-Core develops a core which is able to switch between different execution modes at runtime including a programmable loop-acceleration, out-of-order, asynchronous and the classical RISC/VLWI method[8]. These extensions and architectures address a wider area of usage than database systems. Hardware which is specialized in database applications could enhance the performance and energy efficiency even further.

Commercial Database Systems There have also been commercial approaches to hardware-software co-design for database systems. Oracle introduced *Software in Silicon* in their Spark M7 processors [Or15, Or16]. In detail, three aspects are optimized using custom made hardware: (1) application data integrity, (2) memory decompression, and (3) query processing performance. The first is meant to detect buffer overflows and memory access violations in main memory. Thus, it addresses the memory system layer (b). The second and the third aspect are incorporated in *Database Acceleration Engines*, which decompress data, filter values, and format results, all in one hardware accelerator. A database acceleration engine can be seen as a specialized processing element. This approach needs a hypervisor to queue work to a specific accelerator while the actual CPU cores submit the work asynchronously. On the other hand, our instruction approach extends each single core, which can execute each custom-made instruction in one cycle. Therefore, work is not asynchronously submitted and the core stays in control of the process.

6 Summary and Future Work

In this paper, we have given an overview of our work to support database engines by customizing the Tomahawk processor. In particular, we summarized our database-specific instruction set extensions for the processing elements as well as our iDMA approach for optimizing the main memory access by offloading appropriate logic to the DMA controller. Our experiments are done either using real hardware or using a cycle accurate simulator. Both evaluations show the high potential of our research work regarding performance and energy consumption, especially compared to a general-purpose Intel i7-6500U processor.

Generally, we envision a customized storage and data pre-processing hardware system for database engines. So far, our conducted research work is a first step in this direction. Next, we want to generalize the creation of database-specific instruction set extensions, so that we are able to transfer more database primitives into hardware extensions. After that, we want

[8] Hyperion Core - `http://hyperion-core.com`

to switch the view from a single PE to the complete processor. At this level, the interesting questions are: (i) which PE should contain which instruction set extensions (a single PE cannot offer all extensions due to thermal and space constraints) , (ii) how often does an extension have to be available at all, (iii) which extensions can be best combined, e.g. to reduce the space or co-locate frequently combined extensions. There are a lot of open research questions. Furthermore, the influence on query processing has to be examined, that means, how to do the query processing if we now have a large number of PEs, each offering different functionalities. From our point of view, this introduces a new dimension into query optimization.

Aside from the processing elements, we also want to extend our work on the iDMA controller approach. In addition to point queries, we also want to support range queries for different data structures. Furthermore, we want to investigate what we will be able to outsource to the iDMA, e.g. can we compress or decompress data on the iDMA controller. Nevertheless, this requires balancing our demands of universality and technical feasibility as described in Section 3.1. Furthermore, space and thermal constraints have to be considered as well. Afterwards, the influence on the query processing has to be investigated.

References

[An13] Anati, Ittai; Gueron, Shay; Johnson, Simon; Scarlata, Vincent: Innovative technology for CPU based attestation and sealing. In: HASP. volume 13, 2013.

[APD15] Arulraj, Joy; Pavlo, Andrew; Dulloor, Subramanya R.: Let's Talk About Storage - Recovery Methods for Non-Volatile Memory Database Systems. In: SIGMOD. pp. 707–722, 2015.

[Ar14a] Arnold, Oliver; Haas, Sebastian; Fettweis, Gerhard; Schlegel, Benjamin; Kissinger, Thomas; Karnagel, Tomas; Lehner, Wolfgang: HASHI: An Application Specific Instruction Set Extension for Hashing. In: ADMS@VLDB. pp. 25–33, 2014.

[Ar14b] Arnold, Oliver; Haas, Sebastian; Fettweis, Gerhard; Schlegel, Benjamin; Kissinger, Thomas; Lehner, Wolfgang: An application-specific instruction set for accelerating set-oriented database primitives. In: SIGMOD. pp. 767–778, 2014.

[Ar14c] Arnold, Oliver; Matus, Emil; Noethen, Benedikt; Winter, Markus; Limberg, Torsten; Fettweis, Gerhard: Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs. TECS, 13(3s):107, 2014.

[Bi15] Birrittella, Mark; Debbage, Mark; Huggahalli, Ram; Kunz, James; Lovett, Tom; Rimmer, Todd; Underwood, Keith D.; Zak, Robert C.: Intel Omni-Path Architecture, Enabling Scalable, High Performance Fabrics. Technical report, Intel Corporation, 2015.

[DP10] Deliège, François; Pedersen, Torben Bach: Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In: EDBT. 2010.

[Es11] Esmaeilzadeh, Hadi; Blem, Emily; St. Amant, Renee; Sankaralingam, Karthikeyan; Burger, Doug: Dark Silicon and the End of Multicore Scaling. In: ISCA. 2011.

[Fa12] Farber, Franz; Cha, Sang Kyun; Primsch, Jurgen; Bornhovd, Christof; Sigg, Stefan; Lehner, Wolfgang: SAP HANA Database: Data Management for Modern Business Applications. SIGMOD Rec., 40(4):45–51, 2012.

[Fi08] Firasta, Nadeem; Buxton, Mark; Jinbo, Paula; Nasri, Kaveh; Kuo, Shihjong: Intel AVX: New frontiers in performance improvements and energy efficiency. Intel white paper, 2008.

[Fr11] Francisco, Phil et al.: The Netezza data appliance architecture: A platform for high performance data warehousing and analytics. IBM Redbooks, 3, 2011.

[FSV10] Fusco, Francesco; Stoecklin, Marc Ph.; Vlachos, Michail: NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. PVLDB, 2010.

[Ha16a] Haas, Sebastian; Arnold, Oliver; Nöthen, Benedikt; Scholze, Stefan; Ellguth, Georg; Dixius, Andreas; Höppner, Sebastian; Schiefer, Stefan; Hartmann, Stephan; Henker, Stephan et al.: An MPSoC for energy-efficient database query processing. In: DAC. p. 112, 2016.

[Ha16b] Haas, Sebastian; Arnold, Oliver; Scholze, Stefan; Höppner, Sebastian; Ellguth, Georg; Dixius, Andreas; Ungethüm, Annett; Mier, Eric; Nöthen, Benedikt; Matus, Emil Matus; Schiefer, Stefan; Cederstroem, Love; Pilz, Fabian; Mayr, Christian; Schüffny, René; Lehner, Wolfgang; Fettweis, Gerhard: A Database Accelerator for Energy-Efficient Query Processing and Optimization. In: NORCAS. 2016. to appear.

[Ha16c] Haas, Sebastian; Karnagel, Tomas; Arnold, Oliver; Laux, Erik; Schlegel, Benjamin; Fettweis, Gerhard; Lehner, Wolfgang: HW/SW-Database-CoDesign for Compressed Bitmap Index Processing. In: ASAP. 2016.

[He09] He, Bingsheng; Lu, Mian; Yang, Ke; Fang, Rui; Govindaraju, Naga K.; Luo, Qiong; Sander, Pedro V.: Relational Query Coprocessing on Graphics Processors. ACM Trans. Database Syst., 34(4), 2009.

[In08] Intel Corporation: Intel QickPath Architecture. Technical report, 2008.

[Ke98] Keeton, Kimberly; Patterson, D; Hellerstein, Joseph; Kubiatowicz, John; Yelick, Katherine: The intelligent disk (IDISK): A revolutionary approach to database computing infrastructure. Database, 9(6 S 5), 1998.

[KSL13] Kiefer, Tim; Schlegel, Benjamin; Lehner, Wolfgang: Experimental Evaluation of NUMA Effects on Database Management Systems. In: BTW. pp. 185–204, 2013.

[Me07] Mellanox Technologies: InfiniBand Software and Protocols Enable Seamless Off-the-shelf Applications Deployment. Technical report, 2007.

[MTA09a] Mueller, Rene; Teubner, Jens; Alonso, Gustavo: Data processing on FPGAs. PVLDB, 2(1):910–921, 2009.

[MTA09b] Mueller, Rene; Teubner, Jens; Alonso, Gustavo: Streams on Wires: A Query Compiler for FPGAs. PVLDB, 2(1):229–240, 2009.

[Or15] Oracle Software in Silicon. Oracle Open World Presentation, 2015.

[Or16] Oracle's SPARC T7 and SPARC M7 Server Architecture. White Paper, 2016.

[SG10] SGI® Altix® UV GRU Development Kit Programmer's Guide, 2010.

[We15] Wei, Xingda; Shi, Jiaxin; Chen, Yanzhe; Chen, Rong; Chen, Haibo: Fast In-memory Transaction Processing Using RDMA and HTM. In: SOSP. pp. 87–104, 2015.

[WOS04] Wu, Kesheng; Otoo, Ekow J; Shoshani, Arie: An efficient compression scheme for bitmap indices. Lawrence Berkeley National Laboratory, 2004.

[WTA13] Woods, Louis; Teubner, Jens; Alonso, Gustavo: Less watts, more performance: an intelligent storage engine for data appliances. In: SIGMOD. pp. 1073–1076, 2013.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 403

Hardware-Sensitive Scan Operator Variants for Compiled Selection Pipelines

David Broneske[1], Andreas Meister[1], Gunter Saake[1]

Abstract: The ever-increasing demand for performance on huge data sets forces database systems to tweak the last bit of performance out of their operators. Especially query compiled plans allow for several tuning opportunities that can be applied depending on the query plan and the underlying data. Apart from classical query optimization opportunities, it includes to tune the code using code optimizations for processor specifics, e.g., using Single Instruction Multiple Data processing or predication. In this paper, we examine code optimizations that can be applied for compiled scan pipelines that include aggregations, evaluate impact factors that influence the performance of the scan pipelines, and derive guidelines that a query compiler should implement to choose the best variant for a given query plan and workload.

Keywords: Multi-Column Selection Predicates, Scans, Hardware Sensitivity, SIMD, Predication

1 Introduction

With the advent of main-memory databases, where all necessary data to be processed fits into memory, good performance of database operator code has become a key challenge [Ba13; BHS14]. A superior method to produce fast, optimized code for database queries is query compilation [Ne11]. In essence, a modern query compiler will split a query into several[2] so-called *pipelines* that merge the code of a set of operators into a single loop.

Important ingredients for a pipeline are `if`-clauses, that represent selections. Inside the `if`-clauses, there are further operators, such as projections and hash-table probes and, at last inside the `if`-clause, code for pipeline breakers, such as aggregations or hash-table builds. Having selections in the pipeline, the code becomes prone to branch mispredictions due to the if-clause. In an earlier work, we already experimented with several optimizations of a selection operator [BS14]. Among others, we evaluated the performance of a branching scan against a branch-free (predicated) variant of the scan and also against a scan using the concept of *Single Instruction Multiple Data* (SIMD). The result was, as shown in Figure 1.a, that the predicated variant is overall the best choice. However, these experiments are limited to single predicate selections materialized into a position list; whereas, query compilation allows for several predicates merged into a pipeline including further code inside the loop, for instance, for executing the aggregation. Both scenarios can be found in the TPC-H queries [Tr14], e.g., in query `Q1` and query `Q6`. `Q1` consists of a selection on a single predicate

[1] University of Magdeburg, Database and Software Engineering Group, Universitätsplatz 2, 39106 Magdeburg, firstname.lastname@ovgu.de. This work was partially funded by the DFG (grant no.: SA 465/50-1).

[2] A query may have to be split into several of these pipelines, whenever there is a pipeline breaker (e.g., a join) that needs to consume all the input tuples for the query to proceed.

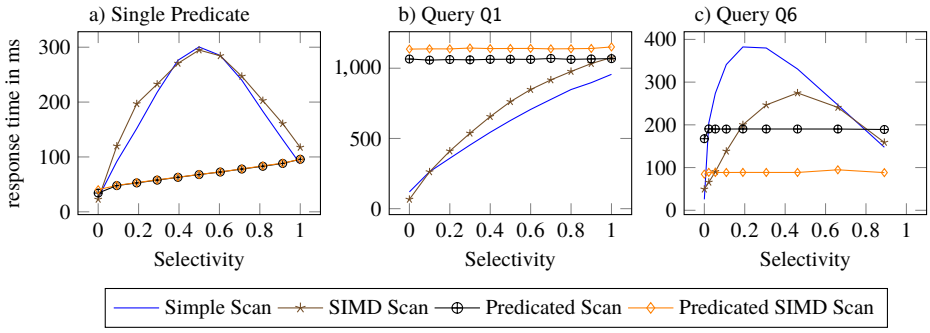


Fig. 1: Response time for different scan implementations on a TPC-H scale factor 10 Lineitem table.

with six aggregates inside the loop, while Q6 has three predicates and a single aggregate. In Figure 1, we depict the runtime of both queries for varying selectivities. In contrast to the single predicate, where the predicated variants are superior, for query Q1 the branching variants are superior, because the overhead of predication that is added to the work inside the loop is too heavy. Furthermore for query Q6, we can see that a SIMD scan is beneficial compared to the normal scan for a range of selectivities when evaluating more than one predicate. Hence, the work inside the loop body and the number of predicates are important characteristics that have to be investigated to find the optimal scan implementation.

In this paper, we contribute a reasonable 3-dimensional evaluation of the impact of *the work inside the loop body*, the impact of *different selectivities for two predicates*, as well as the impact of *the number of predicates* on the performance of a scan pipeline including aggregations. As a result, we will derive guidelines when to use which scan implementation variant to gain the optimal performance.

The remainder of the paper is structured as follows. In Section 2, we present possible code optimizations from the literature that can be applied to a scan operator. Then, we evaluate the resulting scan pipelines using three different scenarios in Section 3. Finally, we present related work in Section 4 and conclude in Section 5.

2 Variants for Predicate Evaluation

There are several ways to optimize a selection operator. In this section, we will review three implementation strategies of single-predicate selections. Furthermore, when considering more than one predicate, we also have to care about how to combine the predicates, which adds the possibility for another variant to be considered for a query compiler. As a side note, all these optimizations are also applicable for code that is not produced by a query compiler. However, implementing one variant per predicate type, number of predicates, and aggregate creates a huge implementation effort and will probably avoided in practice.

2.1 Single-Predicate Variants

There are three implementation strategies for a single-predicate scan. They include to use an `if`-clause, which results in a *branching variant*, to use *predication*, which results in a branch-free variant, and to use *SIMD* to accelerate the predicate evaluation. Furthermore, a SIMD can be implemented using an `if`-clause or predication, which adds another possible variant. Notably, using a multi-threaded implementation is another optimization for scans. However, multi-threading adds a constant speedup to the before-mentioned variants without changing the overall performance differences between the three implementation strategies [BS14].

Branching Scan. The code that is generated for a branching scan is pretty simple. The pipeline itself contains a loop iterating over the input to which the branching scan is adding an `if`-clause with the predicate evaluation. Inside the `if`-clause, subsequent operators will put their code. In our example in Listing 1, we show the branching scan for a `less than` predicate and an aggregation on a specific column that is executed depending on the result of the predicate evaluation. However, this implementation will only benefit from very high selectivities (i.e., only a small amount of tuples match), because else the CPU incurs many branch mispredictions causing considerable performance penalties [BS14; RBZ13].

```
1  for(int i = 0; i < input_size; ++i){
2    if(col[i] < pred)
3        agg+=agg_col[i];
4  }
```

List. 1: Branching scan for `less than` predicate adapted from [BS14].

```
1  for(int i = 0; i < input_size; ++i){
2    agg+=agg_col[i]*(col[i] < pred);
3  }
```

List. 2: Predicated scan for `less than` predicate adapted from [BS14].

```
1  for(int i = 0; i < simd_size; ++i){
2    mask= SIMD_COMP(simd_col[i],pred);
3    if(mask){
4      for (int j=0;j < SIMD_LENGTH;++j){
5        if((mask >> j) & 1)
6          agg+=agg_col[i];
7      }
8    }
9  }
```

List. 3: SIMD scan for `less than` predicate adapted from [BS14].

Predicated Scan. The predicated scan omits an if-clause and will always write a result. For this, we can use in C++ the return value of the comparison to manipulate the written result [RBZ13; Ro04]. The comparison will produce 1, if the condition is true, and 0 if the condition is false. This trick can help for creating an RID list [BS14], but also for manipulating an aggregate computation. For this, you just multiply the aggregate with the result of the predicate, which will mask the aggregate value if the condition is false. We show the resulting code for a predicated aggregating scan in Listing 2.

SIMD Scan. *Single Instruction Multiple Data* is a technology that enables to execute one instruction on several data items in parallel. Since most of the processor's control logic depends on the number of in-flight instructions and registers, but not on the size of the registers, a theoretical speedup of factor n for a vector size of n elements is possible. However, this is hardly achieved in practice. For instance, the SIMD scan cannot benefit from the full potential, as it is impossible to do branching on a single data item in a SIMD fashion. To

perform a selection on a single data item, a mask has to be extracted and depending on the bits that are set, the branching is executed in a scalar fashion [ZR02]. We visualize the code of the branching SIMD scan in Listing 3. The computation of the mask is done in Line 2, that includes a SIMD macro for the predicate evaluation as well as a macro for extracting the mask. Afterwards, the mask is checked for at least one match and from Line 4 to Line 7 the single bits of the mask are examined. In case of a match, the aggregation is executed.

Predicated SIMD Scan. Another option is to fully avoid branches in the SIMD scan. In this variant, we use a bitwise `AND` between the result of the SIMD predicate evaluation and the aggregate column to mask mismatching values of the aggregate column. To get the final result, all entries of the SIMD register have to be summed up at last. Due to space reasons, we refer the interested programmer to the code in our GIT repository[3].

2.2 Multi-Predicate Variants

To evaluate multiple predicates, we just have to extend the above variants to evaluate not one, but several predicates. However, how to combine the predicates in an `if`-clause opens another tuning opportunity, because we can use a conditional `AND` or a bitwise `AND`. Notably, these two variations only apply to the branching scan, because (1) the predicated omits every branch and, thus, will only use the bitwise variant and (2) the SIMD variants only support a bitwise `AND` as macro.

Conditional `AND`. The conditional `AND` (e.g., $pred_1$ && $pred_2$) between predicates will start to evaluate the first predicate at first and only if it evaluates to true, the second predicate will be evaluated. This is also often called a short-circuit evaluation, because the computation of further predicates can be skipped, when the first predicate is already evaluated to false. So, it yields a speedup, if the first predicate is very selective [Ro04]. However, each conditional `AND` will produce a conditional branch in the execution and may lead to heavy branch misprediction penalties.

Bitwise `AND`. The bitwise `AND` (e.g., $pred_1$ & $pred_2$) inside an `if`-clause will evaluate all predicates, form the final result, and then execute the branch according to the result of the predicate evaluation. In this way, branch misprediction penalties are reduced (except the last one for the `if`-clause), but it misses the possibility to skip irrelevant predicates as the conditional `AND` can do.

3 Performance Evaluation of Scan Variants

In this section, we are presenting the performance differences of our scan variants for the three different impact factors that can be derived from the motivational example in Figure 1. The impact factors are the *work* done inside the loop, the *selectivity differences* between several predicates, and also the *number of predicates* to be evaluated. The scans were implemented in C++ and compiled with the GCC 5.1 All experiments are executed on an

[3] http://git.iti.cs.ovgu.de/dbronesk/BTW-Pipeline-Variants

Intel Xeon E5-2630 v3 using a single thread. As a side note, parallelized variants would show the same behavior as the single-threaded variants only with a meaningful speedup. The scanned data is taken from the TPC-H Benchmark and uses the `LineItem`[4] table of scale factor 10. Our predicates are `less then` predicates evaluated on independent columns of the table and predicate values are chosen from the data such that we meet the required selectivities.

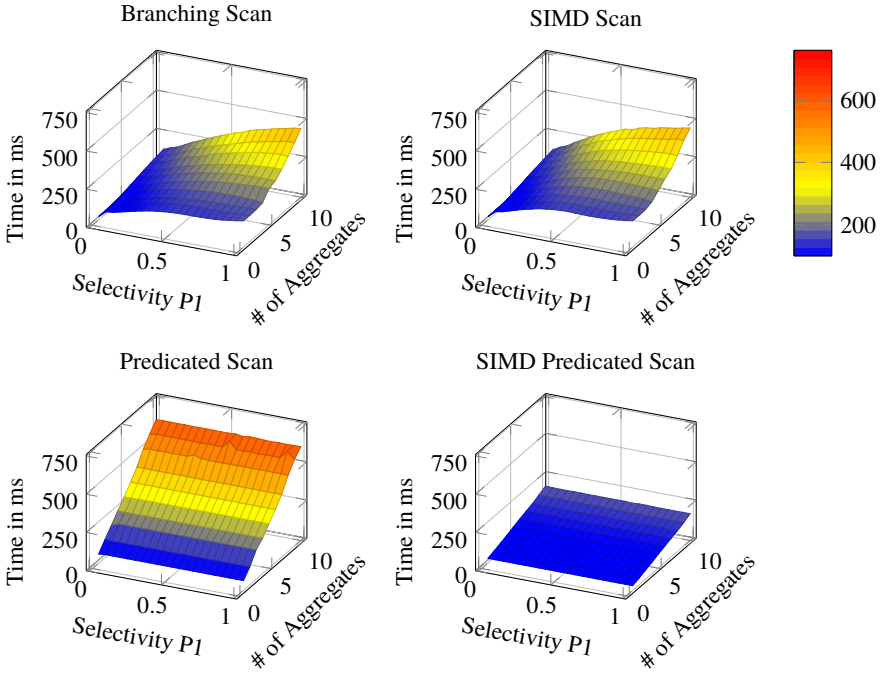


Fig. 2: Variant response time under different numbers of aggregates.

3.1 Single-Predicate Variants under Differing Workload Inside the Loop

For the evaluation of the impact of differing workload inside the loop, we emulate a pipeline with a single predicate and varied the number of aggregates inside the loop. For each aggregate, we have a separate field in the output object and we sum up a unique column per aggregate. Notably, other aggregate functions will only add a different overhead to the execution, but the overall behavior of the variants will stay the same. For instance, the aggregate function `count()` will issue minimal memory access cost (or none), because the aggregate will probably be held in the CPU register and is only incremented. In contrast, holistic aggregate functions (e.g., median) will at first gather all matching values, which will create two memory accesses (for reading the input and for writing the output

[4] The `LineItem` table of scale factor 10 contains 60 million tuples, such that a scan will touch 240 MB of data, which exceeds cache sizes by far. Hence, a bigger scale factor would lead to the same performance behavior.

array). Nevertheless, especially the SIMD predicated version is not applicable for grouped aggregations, as it would use a scatter operation. Hence, this is open for future work.

Since we are only varying the workload inside the loop in this experiment, we only use the four single-predicate variants: branching scan, predicated scan, SIMD scan, and predicated SIMD scan. We visualize the response time of our variants for different numbers of aggregates in Figure 2.

Branching Variants. We can derive from the plots for the branching variants that the selectivity factor has a big impact on the performance only if the number of aggregates is small. The more aggregates we take, the more linear is the progression of the curve. This means, that the overhead for a branch misprediction is hidden by other computations. Overall, the branching scan outperforms the SIMD scan except for selectivity factors around 0, where SIMD has a performance benefit of 13 %.

Predicated Variants. The predicated variants show no impact of the selectivity factor, but a high impact of the number of aggregates, which is consistent with our expectation. However, we can see, that the predicated SIMD scan consistently outperforms the predicated scan by a factor between 2 and 3.5.

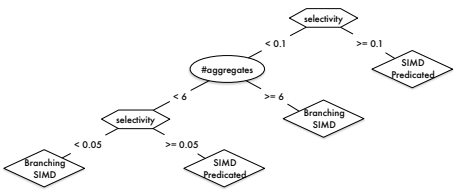


Fig. 3: Decision tree for several aggregates.

Overall Comparison. Comparing the branching variants with the predicated variants, we can see that the selectivity factor range, in which the predicated variants are better than the branching variants, becomes smaller the more aggregates have to be computed. In particular, the branching SIMD scan outperforms the SIMD predicated scan for selectivity factors smaller than 0.05 for up to five aggregates, while the threshold is at 0.1 for more than five aggregates (cf. Figure 3).

3.2 Multi-Predicate Variants for Differing Predicate Selectivities

In this experiment, we want to examine the impact of different selectivities of two predicates on the multi-predicate variants. The evaluated variants are: the branching scan using conditional `AND`, the branching scan using bitwise `AND`, the predicated scan, the SIMD scan, and the predicated SIMD scan. For a good comparison, we use a single aggregation inside the loop, because it is a case, where the predicated variants should still show considerable performance compared to the branching variants. This experiment is especially interesting for the two new variants of the branching variant, because the conditional `AND` is sensitive to the single selectivities, while the other branching variants are only sensitive to the accumulated selectivity. Notably, a query optimizer would order the predicates on their selectivity such that the first predicate would be the most selective. Hence, only the half of the parameter space that we use in this experiment applies for real-world scenarios.

Branching Variants. If we compare the branching variants, we can see that our expectations were partially met. The conditional `AND` scan shows a high performance dependence on the first predicate, while the SIMD branching scan shows a symmetric behavior w.r.t. the order of the predicates (i.e., it does not matter whether P1 is more selective than P2, or the other way around). However, the bitwise `AND` scan still shows a lazy evaluation, which is caused by a bad compiler optimization. Still, if the first predicate has a selectivity factor around 0, the conditional `AND` scan outperforms the other two branching scans by up to factor 1.4. However, if both predicates are very selective reaching a small combined selectivity factor higher than 0.05, the SIMD scan outperforms the conditional `AND` scan.

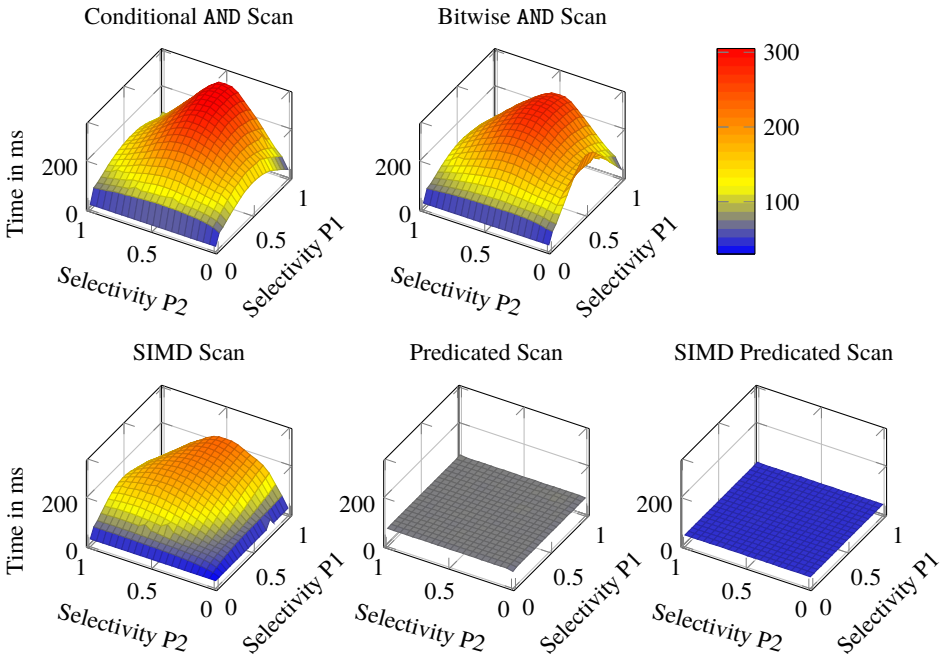


Fig. 4: Variant response time under different selectivities of two predicates.

Predicated Variants. For the predicated variants, we can see that the selectivity factor has no impact on the variant performance, as the resulting performance graph is a plane parallel to the x-y plane. Comparing both predicated variants, the SIMD predicated scan outperforms the predicated scan by a factor of 1.6 in all cases.

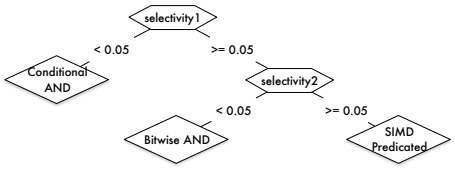


Fig. 5: Decision tree for different selectivities.

Overall Comparison. When comparing branching and predicated variants, we can see that if the predicate with the highest selectivity has a selectivity factor higher than 0.05, the predicated SIMD scan performs better than the branching scans (cf. Figure 5). Else, one of the branching scans performs best depending on the single selectivities.

In summary, the decisions of a query compiler for real-world scenarios depend on the selectivity of the predicate with the highest selectivity due to the predicate ordering during query optimization. If the first predicate has a selectivity close to 0, the conditional `AND` scan will be chosen by the query compiler. In any other case, the SIMD predicated scan will be chosen.

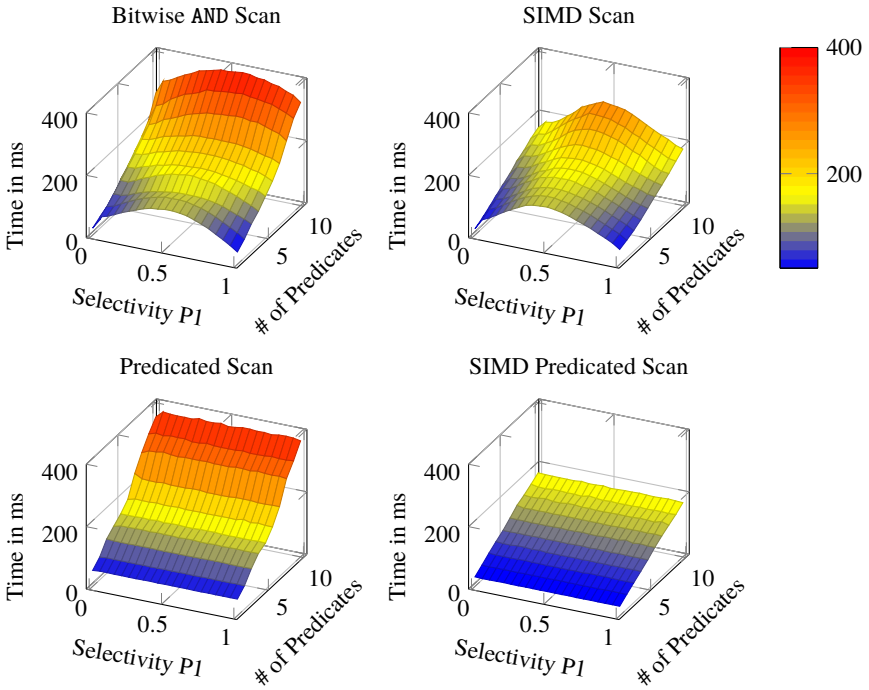


Fig. 6: Variant response time for different numbers of predicates.

3.3 Multi-Predicate Variants for Differing Number of Predicates

In this experiment, we want to examine the impact of an increasing number of predicates on the performance of our multi-predicate variants, especially looking at the difference between SIMD and non-SIMD variants. For the experiment, we exclude the conditional `AND` scan, because we cannot visualize the different selectivities of more than two predicates. In fact, we keep the accumulated selectivity factor ranges constant while stepwise adding more predicates in order to increase the work of predicate evaluation. Similar to the experiment before, we use a single aggregate as work inside the loop.

Branching Variants. Similar to the increase of work inside the loop, an increasing number of predicates will also smoothen the curve of branching scans along the selectivity factor, although this is more evident for the bitwise `AND` scan than for the SIMD scan. This is due to the small overhead of the branch misprediction when compared to the overall response time. Furthermore, when comparing the SIMD scan with the bitwise `AND` scan, we can see that

the bitwise `AND` scan does not scale as good as the SIMD scan with an increasing number of predicates and it can only partially outperform the SIMD scan for a small number of predicates. In fact, for up to four predicates, the bitwise `AND` scan outperforms the SIMD scan for selectivity factors in the range [0.1,0.65]. However for more than four predicates, the SIMD scan constantly outperforms the bitwise `AND` scan, because it can reuse the results inside the SIMD registers.

Predicated Variants. The predicated variants show no impact of the selectivity factor on their performance. However, the number of predicates adds a constant factor to the response time, which is higher for the predicated non-SIMD variant than for the predicated SIMD variant. Similar to the branching variants, for a *single* predicate, the non-SIMD variant performs better than the SIMD variant, while it is the other way around for more than one predicate.

Overall Comparison. When comparing the branching and the predicated variants, we can see that for selectivity factors close to 0, the branching variants outperform their predicated counterparts. For selectivity factors close to 0, performance factors from 1.23 to 1.64 are possible. However, for a selectivity factor between [0.05,0.95], the predicated variants reach a benefit of up to factor of 2.1 in comparison to the best performing branching variant.

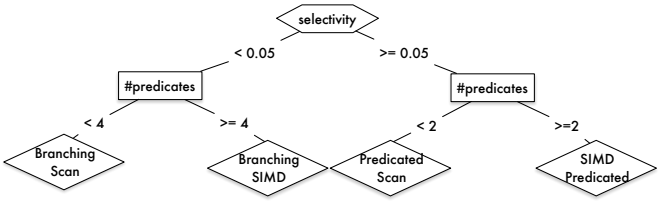


Fig. 7: Decision tree for several predicates.

As shown in Figure 7, a query compiler should use a branching scan for accumulated selectivity factors less than 0.05. In this case, the branching scan is chosen for less than 4 predicates while, otherwise, the SIMD scan is used. For other selectivity factors, the query compiler should use a predicated variant – it should use a predicated SIMD variant for more than one predicate and for one predicate, it should rely on a non-SIMD variant.

4 Related Work

Using SIMD to accelerate database operations, in particular selection conditions, has gained much attention in research. The first idea of a SIMD scan, which is also our basis for the SIMD scan, was proposed by Zhou and Ross [ZR02]. Later on, Polychroniou and Ross extend the work to AVX by using bloom filters [PR14], Sitaridi and Ross to exploit GPUs [SR13] and Polychroniou et al. to exploit Xeon Phis [PRR15]. Furthermore, Willhalm et al. use compression and SIMD acceleration to speed up the scan for single [Wi09] and complex [Wi13] predicates. All these improvements propose reasonable variants that could be used in future work to extend the findings of this paper and could be merged into a cost model for query compilers.

5 Conclusion

In our paper, we compare the performance of different scan pipeline implementations w.r.t. increasing work inside the loop, differing selectivities of two predicates, and increasing numbers of predicates. Our evaluation has shown, that all parameters have a high impact on the usage of the variants and a query compiler should take care of these parameters to select the right variant in order to produce the best performing pipeline code. Nevertheless, exact thresholds still depend on the used machine and have to be re-evaluated for every new CPU architecture.

References

[Ba13] Balkesen, C.; Teubner, J.; Alonso, G.; Özsu, M. T.: Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In: ICDE. IEEE Computer Society, pp. 362–373, 2013.

[BHS14] Broneske, D.; Heimel, S. B. M.; Saake, G.: Toward hardware-sensitive database operations. In: EDBT. Pp. 229–234, 2014.

[BS14] Broneske, D.; Saake, S. B. G.: Database scan variants on modern CPUs: A performance study. In: VLDB Workshop IMDM. Vol. 8921. LNCS, Springer, pp. 97–111, 2014.

[Ne11] Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB 4/9, pp. 539–550, 2011.

[PR14] Polychroniou, O.; Ross, K.: Vectorized bloom filters for advanced SIMD processors. In: SIGMOD Workshop DaMoN. ACM, 2014.

[PRR15] Polychroniou, O.; Raghavan, A.; Ross, K. A.: Rethinking SIMD vectorization for in-memory databases. In: SIGMOD. ACM, pp. 1493–1508, 2015.

[RBZ13] Răducanu, B.; Boncz, P.; Zukowski, M.: Micro adaptivity in Vectorwise. In: SIGMOD. Pp. 1231–1242, 2013.

[Ro04] Ross, K. A.: Selection conditions in main-memory. In: TODS. Vol. 29. 1, pp. 132–161, 2004.

[SR13] Sitaridi, E.; Ross, K.: Optimizing select conditions on GPUs. In: SIGMOD Workshop DaMoN. ACM, 4:1–4:8, 2013.

[Tr14] Transaction Processing Performance Council: TPC BENCHMARK H (Decision Support), tech. rep. 2.17.1, 2014.

[Wi09] Willhalm, T.; Boshmaf, Y.; Plattner, H.; Popovici, N.; Zeier, A.; Schaffner, J.: SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. PVLDB 2/1, pp. 385–394, 2009.

[Wi13] Willhalm, T.; Oukid, I.; Müller, I.; Faerber, F.: Vectorizing database column scans with complex predicates. In: VLDB Workshop ADMS. Pp. 1–12, 2013.

[ZR02] Zhou, J.; Ross, K. A.: Implementing database operations using SIMD instructions. In: SIGMOD. Pp. 145–156, 2002.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 413

DeLorean: A Storage Layer to Analyze Physical Data at Scale

Michael Kußmann[1] Maximilian Berens[1] Ulrich Eitschberger[2] Ayse Kilic[2]
Thomas Lindemann[1] Frank Meier[2] Ramon Niet[2] Margarete Schellenberg[2]
Holger Stevens[2] Julian Wishahi[2] Bernhard Spaan[2] Jens Teubner[1]

Abstract: Modern research in high energy physics depends on the ability to analyse massive volumes of data in short time. In this article, we report on *DeLorean*, which is a new system architecture for high-volume data processing in the domain of particle physics. *DeLorean* combines the simplicity and performance of relational database technology with the massive scalability of modern cloud execution platforms (Apache Drill for that matter). Experiments show a four-fold performance improvement over state-of-the-art solutions.

Keywords: data analysis, big data, LHCb, database, distributed computing, modern hardware, nuclear physics

1 Introduction

As part of the "Big Data" revolution, the way how "data" is being used in applications has seen a dramatic shift over the past years. Increasingly, relevant information is no longer carried in single pieces of data (*e.g.*, data points or records), but comes from the statistical relevance within very large data volumes.

Particle physics is a prime example of this trend that by today has reached virtually any application domain, from science to engineering to business. Analyses over massive sets of experimental data take the role that the close inspection of a single experiment had just a few years ago. To illustrate, the LHCb experiment at CERN's Large Hadron Collider (LHC) produces about 4 terabytes of raw data every second — year-round. Existing systems, even the largest ones, are still overwhelmed by data volumes of this scale.

Large data volumes obviously cry for database technology. Unfortunately, the required analyses are highly complex; scalable database technology lacks the expressiveness to support real-world analyses. As a joint effort with physicist from the LHCb experiment, we therefore developed techniques to bridge the large gap between complex analyses and query capabilities that can provide the necessary efficiency at scale.

In this work, we advocate the use of database technology to accelerate data processing in particle physics. And we report on *DeLorean*, our new, intelligent storage back end to accelerate data analyses at CERN.

[1] TU Dortmund University, Databases and Information Systems Group, {firstname.lastname}@cs.tu-dortmund.de
[2] TU Dortmund University, Experimental Physics V, {firstname.lastname}@tu-dortmund.de

- *DeLorean* builds on *Apache Drill* [Ap16a], the open source counterpart to Google's Dremel system [Me10]. Drill enables relational-style data processing at massive scale, leveraging technologies such as the *Hadoop Distributed File System (HDFS)* or *Apache ZooKeeper* for coordination.
- We show how real-world analysis tasks can be broken up into a *data-intensive part* — leveraging Drill's potential to *scan* massive data volumes in parallel — and into a *compute-intensive part* which covers most of the analysis's complexity but needs to run only on a fraction of the original data set.
- An important ingredient to *DeLorean* is an aggressive reduction of the data volumes that must be *scanned* during the analysis. We achieve this by applying *column-store technology* to a *synopsis* of the original data set, heavily optimized for scanning. In addition, we leverage *lightweight compression* to save bandwidth at the storage layer.
- We illustrate the potential of *DeLorean* using a *reference analysis*, on which we achieve performance improvements of up to a factor 4.6.

We will present *DeLorean* in the following order. Section 2 introduces into particle physics, a prime example for a new class of data-intensive applications. We show physical analyses can be made to scale by separating the data-intensive from the compute-intensive parts in Section 3. In Section 4, we discuss how scans can be optimized through column-oriented designs and compression. We evaluate our proposal on a prototype implementation in Section 5, discuss related work in Section 6, and wrap up in Section 7.

2 The LHCb Experiment at CERN

Particle physics, the *LHCb experiment* at CERN in particular, is an illustrative example of the data-centric nature of modern research in the natural sciences. The greater goal of the LHCb experiment is to understand the *matter-antimatter asymmetry*: why is there more matter in the Universe than antimatter?

2.1 Modern Particle Physics

To find an answer, physicists use the *Large Hadron Collider (LHC)* to accelerate and then collide *proton bunches*. During such collisions, new heavy particles are formed and *decay* shortly after into lighter particles. An example is shown here on the right (decay channel $B^0 \to J/\Psi\, K_s^0 \to \mu^+\mu^- K_s^0 \to \mu^+\mu^-\pi^+\pi^-$).

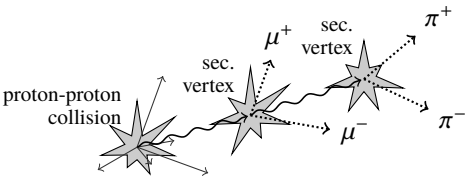


Fig. 1: Example of a collision experiment and its decay products. A B^0 meson is formed during a proton-proton collision, then decays into μ^+/μ^- and π^+/π^- particles.

Of particular interest to physicists are *rare* decay channels. And "rare" has to be taken very literally: probabilities of certain decays range between 10^{-12} and 10^{-15}. Therefore,

physicists produce a massive count of collisions, in the hope to eventually find (some) relevant events.

In practice, collisions are produced at CERN at a rate of up to 40 million collisions per second, year-round. Counting in down times of the accelerator, about 4×10^{14} collision experiments are performed per year, each of which weighs in about 100 kB of data. To handle the resulting massive data volume, data is processed in multiple stages as illustrated in Figure 2. A real-time *trigger system* filters the raw data stream right after data acquisition. Only a small percentage of all data survives this stage and is persisted to a large storage cluster.[3]

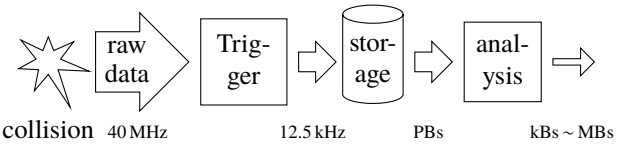


Fig. 2: Processing pipeline at CERN.

The daily task of a physicist is to run analyses on the stored data set. In practice, this means to write a (Python or C++) program that scans over the full data set to extract those collision events that are of interest to the particular question of the physicist. To illustrate, as few as ten $B^0 \to \mu^+\mu^-$ decays were extracted from the entire data set of the first LHC run from 2010 to 2013 [Th15].

2.2 Characteristics of Physical Analyses

Analysis programs typically filter the full, petabyte-scale data set according to complex criteria, strongly dependent on the particular physical question being asked. Thereby, (partial) criteria can be as simple as "return all events that produced a muon particle with an energy of at least . . . ," but also as complex as graph conditions on the 3d tracks that can be inferred — through compute-intensive algorithms — from the data points recorded. Figure 3 here on the right illustrates a strongly simplified example of an analysis that searches for $A^0 \to b^+b^-$ events. As can be seen in the pseudo code, simple predicates (*e.g.* on charge and mass) are interspersed with compute-intensive calculations.

```
for all evt in events do
    for all particle in evt.particles do
        if conditions on tracks and states then ▷ first cut
            calculate information e.g. charge
            if charge < 0 then
                neg ← neg ∪ {particle}
            else if charge > 0 then
                pos ← pos ∪ {particle}
            end if
        end if
    end for
    for all pp in pos, for all np in neg do
        calculate combined mass np.mom. + pp.mom.
        if mass in A^0 mass window then   ▷ second cut
            emit (np, pp)
        end if
    end for
end for
```

Fig. 3: Simplified $A^0 \to b^+b^-$ analysis task (stripping line cut).

[3] Collision experiments filtered out during this stage are lost forever. About 20 PB–30 PB of data are stored to disk every year.

This type of complexity and diversity essentially rules out access structures like (multi-dimensional) indexes, leaving *scans* as the only viable search mechanism.

Analysis performance is, therefore, heavily influenced by the *volume* of the data that is being scanned. To reduce this volume, the existing platform at CERN uses a mechanism that physicists refer to as *stripping*: a *preprocessing stage* segregates all events into *stripping lines* on the storage cluster; each stripping line corresponds to pre-defined search criteria.[4] At this point, few hundred stripping lines are registered in the LHCb system, which was found to be a compromise between selectivity and the cost of pre-processing. In fact, stripping lines need to have a selectivity below 0.5 % to go easy on resources.

The stripping concept is both, a blessing and a curse. While it reduces the scan cost for common analysis (types), stripping *(a)* occupies scarce disk resources and *(b)* is limited to classes of analyses that have been pre-declared to the stripping process. In fact, most physicists would like to get rid of stripping rather sooner than later.

3 Making Analyses Scale

The software frameworks at CERN heavily rely on ROOT [RO16], which — for the context of this work — provides a persistence mechanism for C++ objects. That is, the existing storage layer at CERN consists of serialized C++ objects. For analysis (cf. Figure 2), these objects are de-serialized, then handed over to C++/Python code for processing. To save disk space, ROOT files are aggressively compressed.

The beauty of ROOT is its seamless interplay with the existing analysis code. Over time, a very large library of analysis routines has evolved that is mostly written in C++. It is, however, very difficult to make the approach run efficiently at very large scales, most importantly for two reasons:

(a) C++ object (de)serialization results in relatively complex data structures, on disk as well as in memory. The strategy can, therefore, poorly benefit from modern hardware advances and deep memory hierarchies. As a contrast, relational database engines intentionally stick to a very rigid and well-defined data model — one of the key ingredients to their excellent scalability.

(b) The ROOT de-serialization mechanism will read in C++ objects always as a whole. In practice, many kilobytes have to be read from storage, even when a very simple characteristic (*e.g.* a *charge* value) would be enough to decide that the object can be skipped for a particular analysis.

3.1 *DeLorean*: ROOT + Relational Storage

To avoid — or at least mitigate — the above two problems, *DeLorean* pairs the ROOT framework with a relational storage. On the relational side of *DeLorean*, we store a *synopsis*

[4] A stripping line compares best to a *materialized view* in a relational database engine.

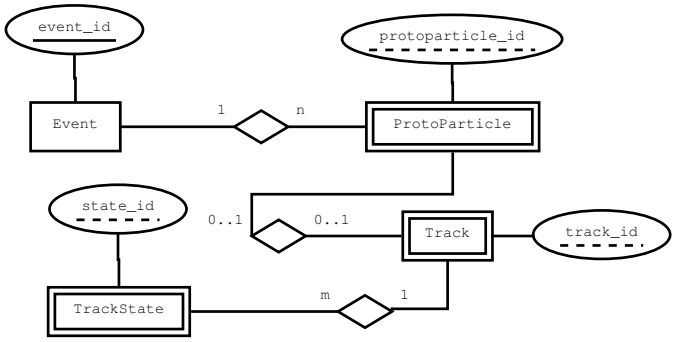


Fig. 4: Data model for the relational part of *DeLorean*.

of the full data set. The synopsis includes those fields of the data set which are queried frequently (using simple, "sargable" predicates) and with high selectivity. With a scan on the relational side, we select *candidate matches*, then de-serialize from the ROOT part only those C++ objects that are still promising.[5] If the relational scan is selective enough, this separation can result in significant savings in the overall scan volume.

Figure 4 visualizes the data model of the relational part of *DeLorean*. In the actual data set, for each collision event about 100 "proto particles" (particles whose kind might not yet be actually certain) and an equal number of particle tracks are being recorded. Over a year, about 10^{11} events are being persisted to the cluster storage. *ProtoParticle*, *Track*, and *TrackState* are factored out from *Event* as weak entities.

Queries over the data set are, again, essentially scans; the joins involved can be answered with efficient merge joins provided that the back-end knows about the physical storage order (by *event_id*). Scans of this type can be parallelized almost straightforwardly. Through (derived) horizontal partitioning, the data set can be distributed efficiently even over very large cloud installations.

In *DeLorean*, we use Apache Drill to realize the relational part. Based on a Hadoop Distributed File System (HDFS), Drill provides a natural way to parallelize typical analysis queries at very large scales. As we shall see in section 4, Drill's column-oriented *Parquet* storage format can assist in optimizing the type of scans typical for *DeLorean*. A big bonus of using a Hadoop-based approach is its fault tolerance which allows to employ cost-efficient consumer hardware.

At this stage of the project, we perform the separation of user analyses into data-intensive and a compute-intensive part (to be ran on the relational and the ROOT side of *DeLorean*, respectively) manually. *DaVinci*, the analysis software framework at CERN, however, already provides a domain-specific language component to express simple predicates over events. As one of our next project steps, we plan to use this language as a basis to automatically extract relational queries in *DeLorean*.

[5] ROOT supports efficient, tree-based seeking to selected events.

4 Optimizing Scans

The stripping line `JPsi2MuMu` in the original system is a good representative for a realistic event selection task. It matches the pattern shown in Figure 3 to select particle combinations where two muons are decay products of a J/Ψ meson. For the stripping line, the two cuts result in selectivities as listed in Table 1. Clearly, the two cuts reflect simple predicates with a very high selectivity (together around 1 : 3,000). As such, they are well-suited to pre-filter events using the relational store of *DeLorean*.

cut	candidates	survivors	selectivity
first	7,325,531	170,858	2.33 %
second	170,858	2,542	1.49 %
total	7,325,531	2,542	0.03 %

Tab. 1: Filter selectivity for the `JPsi2MuMu` stripping line cuts.

4.1 Column-Wise Storage

The characteristics in Table 1 make selection queries in *DeLorean* an excellent candidate to apply *column-store technology*. Storing data in a columnar fashion has two important advantages:

(a) Queries must read from disk only those columns that are actually relevant for the particular filter task (such as *charge* and the *position*-vector).
(b) When a query consists of multiple selection predicates (cuts), data for later attributes must only be fetched from disk for rows where earlier predicates were satisfied. The query optimizer may optimize the order of predicate evaluation accordingly.

Both properties result in a reduction of the data volume that has to be scanned (read from disk) for filtering. For simple queries like the ones we discuss here, a reduction of I/O volume may directly translate into an improved overall performance.

4.2 Lightweight Compression

Column-oriented storage goes well together with *lightweight compression*. With a reduced overall disk memory footprint, the system may be able to read the relevant data from disk with fewer I/O requests and faster. —Such an improvement will, of course, only be beneficial as long as the implied overhead — CPU cost for decompression in particular — does not outweigh the reduction in I/O cost. To this end, earlier work has developed compression schemes that are particularly lightweight and can provide high throughput. The most notable example is the PFOR family of compression schemes of Żukowski *et al.* [Żu06]. In *DeLorean*, we opted for Google's *Snappy* library [Gi16] and gzip, because they integrate particularly well with our base platform Apache Drill.

Traditionally, physicists at CERN have used LZMA for its significantly better compression rates. The high compression rates come at a significant CPU cost for decompression, making

LZMA an inferior choice from a runtime performance perspective. In Section 5, we will report on experimental results to support this claim.

5 Experiments

To test whether Apache Drill is a viable back end option for *DeLorean*, preliminary experiments were conducted to be able to compare DaVinci and *DeLorean*. In this experiment both approaches solely execute the "indexing part" (cf. Figure 3) to show the scalability of *DeLorean*. All experiments have been conducted on a Intel Xeon E5-2609v2 dual socket workstation (8 physical cores, no Hyper-threading) with 64 GB of RAM and a 7200 rpm SATA HDD running Scientific Linux 6.7.

The test data has been provided by the Experimental Physics V group at TU Dortmund University. To be able to run arbitrary indexing tasks, the whole data set is converted to Parquet. Parquet's columnar nature then allows Drill to only process needed columns and only work on a synopsis of the data. Cutting out the stripping process easily outweighs the two-fold data redundancy created by this proceed.

5.1 Compression Algorithms

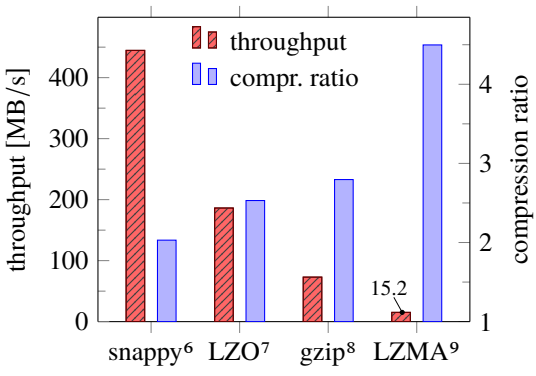


Fig. 5: Comparison of throughput vs. compression ratio for different algorithms on our data set.

Measurements using Intel VTune show that DaVinci spends 50 % of its runtime decompressing data using LZMA. Searching for a more CPU efficient compression algorithm, a small survey brought up three promising candidates: snappy, LZO and gzip. In an experiment we applied four different algorithms to uncompressed *Parquet* files containing actual LHCb data. Figure 5 compares the compression ratios for our data and shows the decompression throughput achieved on our test system (single threaded). LZO looks like a very promising candidate, compromising fairly good on decompression throughput and compression ratio. Unfortunately, Drill does not support LZO compressed Parquet files by now. Further experiments will stick to snappy and gzip for now.

[6] snappy-java 1.0.5.4 (`https://github.com/xerial/snappy-java`)
[7] lzop 1.03 (library version: 2.06) (`http://www.oberhumer.com/opensource/lzo/`)
[8] Oracle GZIPInputStream (`https://docs.oracle.com/javase/7/docs/api/java/util/zip/GZIPInputStream.html`)
[9] LZMA SDK 9.22 (`http://www.7-zip.de/sdk.html`)

Snappy exceeds the common HDD bandwidth by far, but has the lowest compression ratio in the test field and is therefore not a very good compromise for our test system. In section 5.2 we will see that *DeLorean* is in fact IO bound, therefore investigating algorithms with higher compression rates or specialized algorithms may be worthwhile.

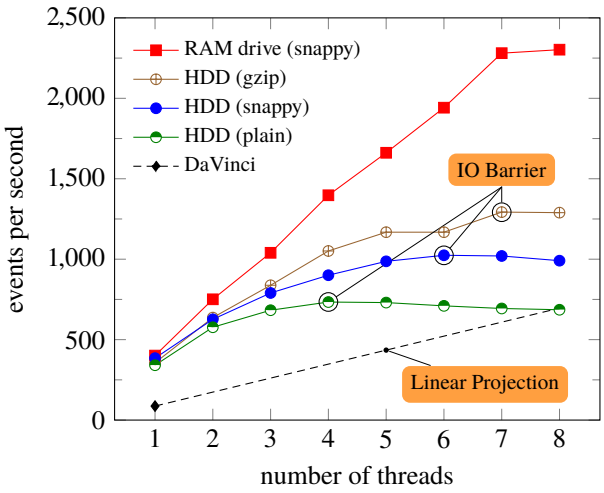


Fig. 6: Single node scalability of *DeLorean* compared to DaVinci.

5.2 Benchmarking *DeLorean*

Using an out-of-the-box embedded Drill instance we were able to achieve an event throughput increase of up to factor 4.6 (single-threaded). Unfortunately, a multi-threaded configuration of DaVinci is currently not available in our laboratory setting. For the sake of fairness we assume a linear scalability for DaVinci (best case). Figure 6 shows the scalability for different compression algorithms compared to the DaVinci projection. One notable result of the experiment is that *DeLorean* even outperforms the linear projection of DaVinci.

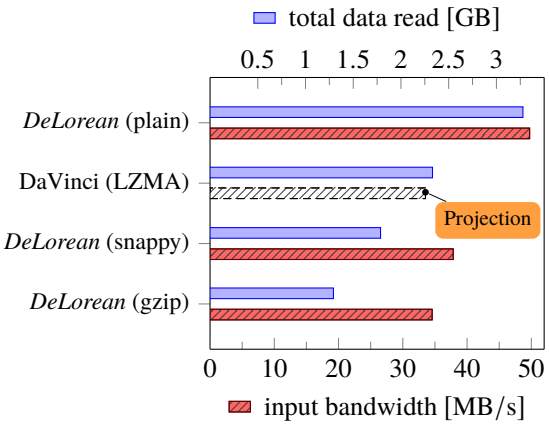


Fig. 7: Comparison of HDD traffic.

Figure 6 clearly shows that the HDD bound experiments hit the IO barrier.[10] Here we can see that compression can leverage the IO bottleneck significantly. Gzip, the "heaviest" of the compression algorithms, performs best in this scenario, so it might be worthwhile to invest in higher compression ratios using domain specific compression algorithms (like run-length encoding or delta encoding).

Talking about IO bottlenecks one should also look at the IO behaviour. Figure 7 shows the total amount of data read by the different approaches and the corresponding throughput at the IO barrier. Although DaVinci uses a very heavy weight compression algorithm, total reading IO is higher than for *DeLorean* using lighter weight algorithms. Due to the columnar storage design, *DeLorean* avoids reading irrelevant

[10] The RAM drive experiment is there to verify the IO bottleneck: Regarding the massive amount of data, it is unrealistic to process the whole data set in memory.

data from disk. Additionally, DaVinci spends about 50 % of its total runtime decompressing data that will never be touched later. Assuming the compression ratio of LZMA from Figure 5, DaVinci processes about three times more uncompressed data than *DeLorean*. Looking at the IO throughput in Figure 7, it becomes clear that the limiting factor cannot be the bandwidth but must be a bad access pattern.[11] We are currently working on strategies to enhance the access pattern by using intelligent caches and exploiting data inherent sortedness (our extracted tables are inherently sorted by ID). Additionally, we expect a lesser impact of access patterns when further scaling *DeLorean*, allowing for much bigger block sizes per thread.

6 Related Work

Notable related work has been done by Duggan *et al.* (*BigDAWG* [Du15]) in optimizing query execution over different storage systems with various data models. *DeLorean* is an example of what Duggan *et al.* call a *polystore system*: A polystore system uses multiple data sources with different data models (in our case relational Parquet files and object-oriented ROOT files) and takes into account all sources when optimizing queries.

There exist a number of competing Big Data platforms such as Apache Pig [Ap16b] and Impala [Ko15]. Both these platforms could serve as a back end for *DeLorean*. Notably, Apache Pig features LZO compression for Parquet files.

Nieke *et al.* [Ni15] analysed the CERN infrastructure with focus on the ATLAS project. They suggest using Apache Hive [Th09] in conjunction with Parquet to speed up data processing in CERN's datacentres.

The complexity of physical analyses rules out the use of common tree-based indexing mechanisms for acceleration, an observation that bears many similarities with the conclusions of Weber *et al.* [WSB98]. As a remedy for inefficient indices in high-dimensional spaces, they propose *VA-files*. As a compact data synopsis, VA-files allow for very fast scans. They are used to pre-filter data much like the column store of *DeLorean*.

Ekanayake *et al.* [EPF08] applied MapReduce-style processing (using Hadoop, among other implementations) to high energy physics data. Their experiments suggest good scalability for said analyses.

7 Summary

With our experiments we showed some weak spots of the current software setup at LHCb: Highly selective scans in conjunction with heavy weight compression. We showed that using (CPU-)balanced compression is essential for achieving high processing throughput.

Introducing *DeLorean*, a polystore system that uses modern database techniques in conjunction with columnar storage we were able to speed up data intensive processing steps in

[11] Common HDDs exceed 150 MB/s reading throughput for sequential reads.

DaVinci. *DeLorean* will bridge the gap between "flat" ROOT files and the relational database world, exploiting the advantages of both worlds, allowing for higher scan throughput and less data redundancy.

Acknowledgements: This work has been supported by the DFG, Collaborative Research Center SFB 876, project C5 (`http://sfb876.tu-dortmund.de/`).

References

[Ap16a] Apache Drill - Schema-free SQL for Hadoop, NoSQL and Cloud Storage, `https://drill.apache.org`, 15.09.16.

[Ap16b] Apache Pig, `https://pig.apache.org`, 20.09.16.

[Du15] Duggan, Jennie; Elmore, Aaron J; Stonebraker, Michael; Balazinska, Magda; Howe, Bill; Kepner, Jeremy; Madden, Sam; Maier, David; Mattson, Tim; Zdonik, Stan: The bigdawg polystore system. ACM Sigmod Record, 44(2):11–16, 2015.

[EPF08] Ekanayake, J.; Pallickara, S.; Fox, G.: MapReduce for Data Intensive Scientific Analyses. In: 2008 IEEE Fourth International Conference on eScience. pp. 277–284, December 2008.

[Gi16] GitHub - google/snappy: A fast compressor/decompressor, `https://github.com/google/snappy/`, 09.09.2016.

[Ko15] Kornacker, Marcel; Behm, Alexander; Bittorf, Victor; Bobrovytsky, Taras; Ching, Casey; Choi, Alan; Erickson, Justin; Grund, Martin; Hecht, Daniel; Jacobs, Matthew; Joshi, Ishaan; Kuff, Lenni; Kumar, Dileep; Leblang, Alex; Li, Nong; Pandis, Ippokratis; Robinson, Henry; Rorke, David; Rus, Silvius; Russell, John; Tsirogiannis, Dimitris; Wanderman-Milne, Skye; Yoder, Michael: Impala: A Modern, Open-Source SQL Engine for Hadoop. In: CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings. www.cidrdb.org, 2015.

[Me10] Melnik, Sergey; Gubarev, Andrey; Long, Jing Jing; Romer, Geoffrey; Shivakumar, Shiva; Tolton, Matt; Vassilakis, Theo: Dremel: Interactive Analysis of Web-scale Datasets. Proc. VLDB Endow., 3(1-2):330–339, September 2010.

[Ni15] Nieke, C; Lassnig, M; Menichetti, L; Motesnitsalis, E; Duellmann, D: Analysis of CERN computing infrastructure and monitoring data. In: Journal of Physics: Conference Series. volume 664, p. 052029, 2015.

[RO16] ROOT a Data analysis Framework, `https://root.cern.ch`, 19.09.16.

[Th09] Thusoo, Ashish; Sarma, Joydeep Sen; Jain, Namit; Shao, Zheng; Chakka, Prasad; Anthony, Suresh; Liu, Hao; Wyckoff, Pete; Murthy, Raghotham: Hive: A Warehousing Solution over a Map-reduce Framework. Proc. VLDB Endow., 2(2):1626–1629, August 2009.

[Th15] The CMS and LHCb collaborations: Observation of the Rare $B_s^0 \to \mu^+\mu^-$ Decay from the Combined Analysis of CMS and LHCb Data. Nature, 522:68–72, June 2015.

[WSB98] Weber, Roger; Schek, Hans-J; Blott, Stephen: A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In: 24th VLDB Conference. New York, USA, pp. 194–205, 1998.

[Żu06] Żukowski, Marcin; Héman, Sándor; Nes, Niels; Boncz, Peter A.: Super-Scalar RAM-CPU Cache Compression. In: 22nd International Conference on Data Engineering (ICDE'06). p. 59, April 2006.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 423

Efficient Storage and Analysis of Genome Data in Databases

Sebastian Dorok[1] Sebastian Breß[2] Jens Teubner[3] Horstfried Läpple[4] Gunter Saake[5]
Volker Markl[6]

Abstract: Genome-analysis enables researchers to detect mutations within genomes and deduce their consequences. Researchers need reliable analysis platforms to ensure reproducible and comprehensive analysis results. Database systems provide vital support to implement the required sustainable procedures. Nevertheless, they are not used throughout the complete genome-analysis process, because (1) database systems suffer from high storage overhead for genome data and (2) they introduce overhead during domain-specific analysis. To overcome these limitations, we integrate genome-specific compression into database systems using a specialized database schema. Thus, we can reduce the storage overhead to 30%. Moreover, we can exploit genome-data characteristics during query processing allowing us to analyze real-world data sets up to five times faster than specialized analysis tools and eight times faster than a straightforward database approach.

Keywords: main-memory database systems, genome analysis, variant calling

1 Introduction

Genome sequencing and analysis promises to detect, predict and prevent diseases based on genetic variations more efficiently than traditional medicine can do [Br13]. Due to next-generation sequencing techniques, genome sequencing becomes cheaper and faster [Li12]. For that reason, reading genome sequences via sequencing machines is not the bottleneck anymore, but the management, analysis and assessment of large amounts of genome data, i.e. *detecting genetic variations* and investigating their consequences [Ma10].

To detect genetic variations, researchers use specialized tools. To investigate the consequences of potential variations, they use database systems that allow for convenient integration with other data sources [Ku07; Le06; Sh05; Tö08]. This separation introduces additional and partly manual effort to ensure reproducibility of results [Sa13]. Avoiding this separation enables researchers to analyze genomes completely within the database system. As a consequence, we can declaratively analyze genome data and improve the comprehensibility of analysis results [RB09]. Furthermore, database systems are able to provide comprehensive data-management features, such as provenance tracking [EOA07] or annotation management [Bh04], that would be available throughout the complete genome analysis process.

[1] Bayer Business Services GmbH & University Magdeburg (Work was done in part when employed at Bayer Pharma AG), sebastian.dorok@ovgu.de
[2] DFKI GmbH (Work was done in part when employed at TU Dortmund), sebastian.bress@dfki.de
[3] TU Dortmund, jens.teubner@tu-dortmund.de
[4] Bayer HealthCare AG, lapple@alumni.stanford.edu
[5] University Magdeburg, gunter.saake@ovgu.de
[6] TU Berlin, volker.markl@tu-berlin.de

Our experiments on integrating genome-analysis tasks, such as detecting genetic variation, into a database system (DBS) demonstrate that we can achieve competitive runtime performance compared to specialized analysis tools. However, DBSs lack appropriate light-weight compression techniques for genome data. For that reason, storage consumption increases by more than a factor of two compared to state-of-the-art flat files, because genome data consists mostly of unique strings that are hard to compress using standard light-weight compression schemes. Additionally, unfavorable string conversions during genome data processing increase time to knowledge within a DBS. Common genome-data encodings represent necessary analysis information implicitly within strings. Considering the required string manipulations, DBSs usually cannot keep pace with specialized analysis tools. To overcome both issues, we exploit genome-specific data and query characteristics within DBSs leading to a database-native application design. We make the following contributions:

1. We **identify genome-data related bottlenecks** in DBSs by performing an in-depth analysis of a straightforward database-approach for variant detection. As optimization targets, we identify missing genome-specific compression schemes and on-the-fly string conversion of genome data via user-defined functions (UDFs).
2. We enable **light-weight genome-specific compression** for DBSs. To this end, we use a specialized database schema allowing us to integrate genome-specific compression. At the same time, it allows us to perform string conversions once during data import, which improves analysis runtime by up to a factor of 1.5.
3. We propose a **genome-specific filtering technique** called *base pruning*. *Base pruning* leverages the characteristic of genome data to be very similar to a given reference genome reducing the number of genome positions to be processed and improving runtime by up to a factor of 5.
4. We **combine genome-specific compression and query optimization** to improve overall performance. Therefore, we outline how we can leverage reference-based compressed data to reduce the runtime of *base pruning*. Moreover, we explain why heavy-weight compression limits the overall benefit of *base pruning*.

Compared to state-of-the-art flat file formats, our techniques can reduce the storage overhead of a DBS approach to 30% without using heavy-weight compression. At the same time, we can detect genetic variation within whole genomes up to five times faster than state-of-the-art analysis tools and up to eight times faster than a straightforward DBS approach.

The remainder of the paper is structured as follows. In Section 2, we introduce the basics of variant detection. In Section 3, we assess a straightforward database-approach for variant detection regarding storage consumption and analysis performance to identify optimization targets. In Section 4, we introduce genome-specific compression schemes for database systems. In Section 5, we explain *base pruning*. In Section 6, we evaluate storage consumption and query performance of our approaches using three real world data sets. In Section 7, we provide an overview of related work.

2 Detecting genetic variation

In this section, we outline the basics of variant detection. First, we motivate the need for variant detection. Then, we describe an important variant detection approach: *Single Nucleotide Variant (SNV)* calling. Finally, we explain a common genome data encoding.

2.1 DNA sequencing and read mapping

DNA molecules encode genetic information via sequences of the four (nucleo)bases Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). DNA sequencing machines make this genetic information digitally readable. To this end, they "read" the sequence of bases within DNA molecules and generate sequences of the characters A, C, G, and T. Therefore, the generated sequences are called *reads*. DNA sequencing techniques are not capable to process complete DNA molecules, but small parts of them only [Qu12]. Thus, in order to reconstruct the sample's complete genome, reads must be assembled.

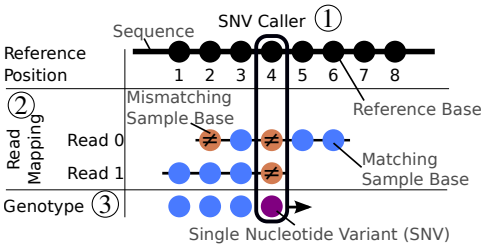


Fig. 1: From mapped reads to SNVs: A SNV caller ① aggregates bases of mapped reads ② per genome position, derives a genotype ③ and calls a SNV if genotype and reference base differ.

A common technique to assemble reads is read mapping [LH10]. Read mapping tools leverage already known reference sequences to reconstruct the sample's genome by mapping reads to the best matching position. In Figure 1, we depict the mapping of two reads (chains of colored circles) to a reference sequence (chain of black circles). Read mapping is a challenging task and has to cope with several difficulties such as deletions, insertions and mismatches (cf. circles with ≠ symbol). These variances can be real variations but also DNA sequencing errors. Therefore, every base in a read has an associated quality value indicating the probability that the base is wrong. In this work, we refer to the output of read mappers using the term *genome data*.

2.2 Variant calling

Usually, scientists are interested in genome sites that differ from a given reference used during read mapping. Such genome sites are called *variants*. The process of detecting variants is called *variant calling*, i.e. determining whether a variant is present or not based on the mapped reads and associated quality information [Ni11]. A special class of variants are SNVs, i.e. differing genotypes at single genome positions (cf. purple circle in Figure 1). The detection of SNVs plays a vital role in genome analysis, because these are known to trigger diseases such as cancer [Ma13]. According to Nielsen et al., two general approaches for SNV calling exist: (1) frequency approaches with fixed cut-off rules and (2) probabilistic approaches incorporating uncertainty in data due to base call and read mapping errors [Ni11]. Independent of the concrete SNV calling approach, the general idea is to aggregate all bases that are mapped to a specific genome position. We depict this idea in Figure 1. The SNV caller (black box) consumes all bases that are mapped to the same genome position and computes a genotype. Afterwards, the SNV caller compares the genotype with the corresponding reference base. In case of a difference, it calls a SNV.

2.3 Encoding of reads

Existing flat-file formats for genome data are optimized to reduce storage consumption. Thus, they encode much information implicitly. For example, the *Sequence Alignment/*

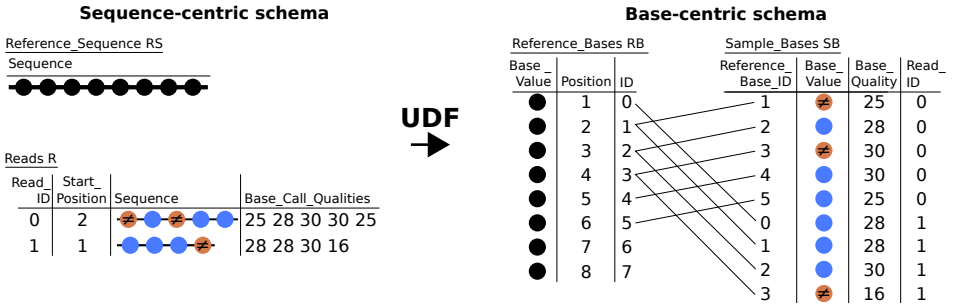


Fig. 2: A sequence-centric schema implicitly models read mapping information similar to flat files. In contrast, a base-centric schema makes mapping information explicit allowing for direct processing.

Map (SAM) format encodes the actual mapping of a read to a given reference as triple of a *Start position*, *DNA sequence* and *CIGAR string* [SA15]. The CIGAR string encodes whether a specific base within the DNA sequence is deleted, inserted, mismatched or matched and, thus, has impact on the actual position of the base. Thus, before performing SNV calling, position information of bases must be made explicit.

3 A straightforward database approach for SNV calling

A straightforward approach for SNV calling using database systems is to store reads similar to the SAM format and to process them similar to specialized analysis tools such as SAMTOOLS [Li09]. In the left part of Figure 2, we depict the basic idea. For every read in table `Reads (R)`, we store the sequence of bases (`R.Sequence`) together with the CIGAR string[7], the corresponding base call quality values (`R.Base_Call_Qualities`) and the start position (`R.Start_Position`). In table `Reference_Sequence (RS)`, we store the reference sequence as string (`RS.Sequence`). We call this data representation *sequence-centric database schema*.

To perform SNV calling, reads must be converted to get explicit access to all bases mapped to a specific genome position (cf. Section 2.3). For example, the analysis tool SAMTOOLS [Li09] transforms mapped reads into an intermediate data structure called *pileup*. A *pileup* lists all bases that map to a specific genome position. Then, SAMTOOLS computes genotypes by aggregating the bases in a *pileup*. We emulate this approach in a database-native way relying on standard database operators where possible. First, we convert the data via a UDF into an intermediate base-centric data representation [DBS14] making read mapping information explicitly available as shown in the right part of Figure 2. The reference sequence and the mapped reads are split into single bases by storing them, literally spoken, vertically in tables `Reference_Bases (RB)` and `Sample_Bases (SB)`[8]. Using a foreign-key relationship (cf. `SB.Reference_Base_ID`), we can explicitly encode which sample base belongs to which reference base. Further information such as read containment (`SB.Read_ID`), position within the genome (`RB.Position`) and base call qualities (`SB.Base_Quality`) is stored in adjacent columns. The explicit mapping information allows us to process genome data

[7] For simplicity, we only consider mismatching bases and omit inserted or deleted bases.
[8] Using the base-centric database schema, we already apply CIGAR operations to the base values of reads.

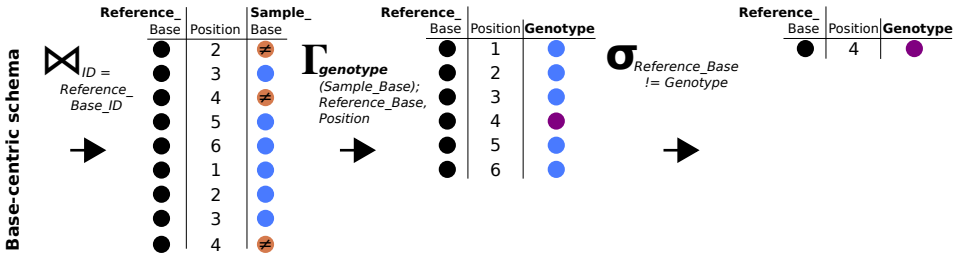


Fig. 3: The base-centric schema allows for direct access and processing of mapped reads via relational database operators.

as shown in Figure 3 using relational database operators. To process a genome region of interest, we join the related bases and aggregate them by genome position using a domain-specific aggregation function called *genotype*. Finally, we filter genotypes that differ from the reference. In the example, we found position four to be a SNV.

In the following, we assess the straightforward approach regarding storage consumption and SNV calling performance to identify advantages and disadvantages. Of course, we have to distinguish between the logical data representation and the physical one. Without loss of generality, we assume that the physical data representation resembles the logical one. As evaluation system, we use CoGaDB [BFT16], a main memory database system, which stores and processes data column oriented similar to MonetDB. Furthermore, we chose CoGaDB, because it provides light-weight compression techniques that we use as baseline for our proposed optimizations. We use a complete human genome provided by the 1000 genomes project that comprises ca. 14 billion mapped sample bases to process. More information on the used data set and the evaluation machine can be found in Section 6.

3.1 Storage consumption

Efficiently storing genome data is hard to achieve as it mainly consists of unique strings. For example, reads are mostly unique to prevent a possible bias within analysis results [De11]. Thus, light-weight string compression schemes such as dictionary encoding increase storage size rather than compressing data. Therefore, we do not compress `RS.Sequence` and `R.Sequence` in a straightforward DBS approach. In Figure 4, we show the results of the DBS approach and the compressed SAM formats CRAM [CR15] and BAM [SA15].

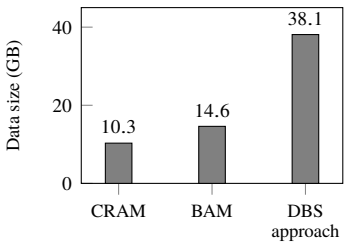


Fig. 4: A DBS approach requires three to four times more storage, which is due to missing compression capabilities.

Using the DBS approach, we require 3.7 times more storage space compared to CRAM and 2.6 times more storage compared to BAM. This is mainly due to the limited compression capabilities for unique strings. Both flat-file formats use heavy-weight compression such as BGZF encoding. CRAM additionally applies reference-based compression [Hs11].

3.2 SNV calling runtime

Now, we compare the SNV calling runtime of the DBS approach with the analysis tool SAMTOOLS [Li09]. We show the runtime results in Figure 5. The DBS approach uses a preloaded database. SAMTOOLS accesses flat files stored in a ramdisk. To make the comparison fair, we compare only the runtime for detecting SNVs and do not consider post-processing validations that can be applied to both approaches. Moreover, we parallelized SAMTOOLS to use all available threads, because SAMTOOLS does not provide such an option natively.

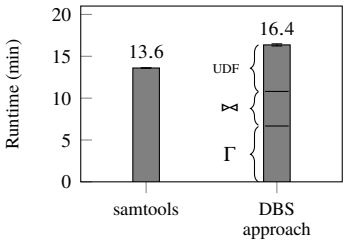


Fig. 5: SNV calling runtime using a DBS approach is comparable to the highly optimized SAMTOOLS.

Considering the overall SNV calling runtime, the DBS approach can be competitive to SAMTOOLS. The read conversion and aggregation phase dominate the runtime. We can speedup the aggregation phase using heuristics to reduce the number of groups to be processed. To speedup the conversion phase, we can integrate domain-specific processing mechanisms similar to specialized analysis tools such as SAMTOOLS. These tools rely on stream processing of genome data and require reads to be sorted by starting position to guarantee low response time. The sorting allows them to interleave data loading with conversion and aggregation, because compressed data-blocks read from disk contain reads of the same genome range. However, we should avoid such black-box behavior within a DBS, because it limits the transparency and portability and is hard to parallelize [RB09].

3.3 Wanted: Genome-specific extensions

Considering the results of the storage and SNV calling runtime experiments, a straightforward DBS approach suffers from missing genome-specific storage and processing capabilities. To reduce storage consumption, we focus on integrating reference-based compression, which achieves good compression ratios [Hs11]. Our goal is to integrate reference-based compression in a light-weight manner to avoid decompression overhead. To improve analysis performance, we want to provide a data layout that avoids string conversions, because these are non-relational operations that are hard to optimize by the DBS. In the following section, we explain how we achieve both goals.

4 Integrating genome-specific compression

State-of-the-art flat-file formats such as CRAM [CR15] use heavy-weight and genome-specific compression schemes to achieve good compression ratios of genome data. Disk-based database systems can hide the decompression overhead of heavy-weight compression when loading data from disk. In contrast, if we use heavy-weight compression in a main-memory database system, we would sacrifice the performance potentials gained from main-memory storage of data. Therefore, we aim to integrate genome-specific compression schemes in a light-weight manner into a DBS reducing decompression overhead by allowing us to process compressed genome data and to decompress single data items fast.

4.1 Light-weight reference-based compression

In the following, we explain how we integrate light-weight reference-based compression into a column-oriented database system. We provide an example in Figure 6.

Reference-based compression [Hs11] is a genome-specific compression scheme. It exploits read mapping information. Usually, the mapped reads and the reference sequence match to a high degree. Thus, the idea is to encode the mapped reads according to the reference sequence. Therefore, we need information about which sample base of a read maps to which reference base of the reference sequence. In a sequence-centric database schema where we store reads as strings, the required information is only given implicitly. This leads to additional overhead when (de-)compressing the data, because we have to extract necessary information before we can use it. What we essentially need is a mapping between sample and reference bases. If we introduce this mapping, we logically end up with a base-centric data representation. Consequently, our idea is to use the base-centric database schema, that encodes the mapping via a foreign-key relationship (cf. Figure 2), as primary data representation to integrate light-weight reference-based compression. At the same time, we remove the overhead of data conversion for SNV calling, if we can store genome data directly using the base-centric database schema.

Concept. The base-centric schema stores the mapping between sample base and reference bases explicitly via the `SB.Reference_Base_ID` column. We can use the column values as index to the `RB.Base_Value` column. Thus, we can look up the reference base to which a sample base is mapped and check whether it is different or not. Instead of storing each base value of a sample, we only store those bases in an *exception* list that are different from their respective reference base. Furthermore, we use a bitmap to mark the unequal bases. In case of good mapping quality, we do not have to store all sample bases. To make this technique efficient, we assume that the reference genome fits into main memory. Thus, we can retrieve sample bases by their row id as follows:

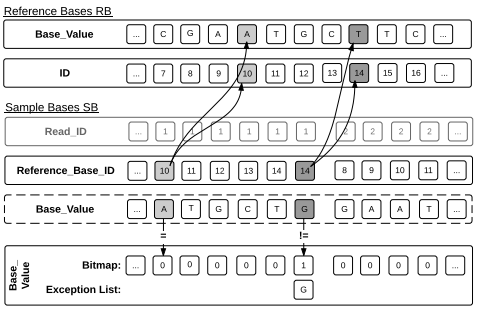


Fig. 6: Reference-based compression uses the existing foreign key relationship to compress sample bases. Differing sample bases are marked and stored in an exception list.

1. Given a row id, check whether the value is different according to the reference by scanning the bitmap.

2.1 In case of difference, use the prefix sum over the bitmap as index to look up the exception value.

2.2 In case of a match, look up the base in column `RB.Base_Value`. As look up index we can use the respective `SB.Reference_Base_ID`.

Improving storage consumption. Nevertheless, this approach requires to store one bit for every sample base. Consequently, if we store all sample bases of the data set used in the previous section, we require ca. 14 billion bits that is ca. 1.75 GB. To further reduce the data size, we use a compressing word-aligned hybrid (WAH) bitmap instead of a plain

bitmap [WOS06]. In a WAHBitmap, zeroes and ones are organized in words of a specific size, e.g. 32 bit. If a word contains only zeroes or ones, it is converted into a fill word encoding the number of words that contain only zeroes or ones. Thus, long runs of zeroes or ones can be compressed effectively. Assuming a reasonable read mapping quality, our bitmap contains many zeroes.

Fast data access. The WAHBitmap applies run-length encoding on bitmaps. Consequently, random access performance is an issue as count values have to be summed up in order to determine the value of a given row. Thus, we extend the WAHBitmap to store the first row id that is not within a word similar as suggested by Abadi et al. for run-length compressed columns [AMF06]. Then, we can use binary search to access random row ids faster.

Since scientists are interested in consecutive genome ranges such as genes or chromosomes, it is quite common that our SNV calling query processes complete reads that are mapped to the specific genome region. Therefore, the chances to access consecutive values are high. Thus, we integrate a mechanism to speed up sequential row accesses by caching the last accessed index and visited word avoiding binary searches.

4.2 Delta+RLE encoding

On the one hand, the base-centric database schema enables us to integrate genome-specific compression in a light-weight way. On the other hand, the base-centric database schema has the drawback to increase the data volume due to explicit encoding of information. In Table 1, we breakdown the storage requirements for single columns of the base-centric database schema when storing the complete human genome data set from Section 3. The breakdown reveals that explicit position information stored in `SB.Reference_Base_ID` and `RB.Position` lead to a massive storage blow up. This information is usually implicitly stored within strings. Consequently, if we want to use the base-centric database schema as primary data representation, we have to cope with the additional storage overhead.

Table	Column	GB	%
Sample_Bases SB	Reference_Base_ID	111.3	72.8
Reference_Bases RB	Position	12.5	8.2
Sample_Bases SB	Base_Value	5.2	3.4
Other columns		23.9	15.6

Tab. 1: Storage breakdown of a human genome using a base-centric database schema. Explicit position information stored in `SB.Reference_Base_ID` and `RB.Position` lead to a large storage blow up.

Both problematic columns contain runs of consecutive values that are incremented by one. This circumstance is inherent to the data as we store the base values of every read consecutively. The `SB.Reference_Base_ID` values are foreign keys to the `Reference_Bases` table. Within a mapped read, it is a common case that consecutive bases are mapped to consecutive reference bases. As we can guarantee that all reference bases are sorted according to their `RB.Position`, `SB.Reference_Base_ID` values are usually incremented by one within the same read. Thus, the delta between two values in one run is mostly one. In the following, we describe an encoding and compression scheme that combines delta encoding and run-length encoding (RLE) to compress such data: *Delta+RLE* encoding.

Concept. In Figure 7, we depict the idea of Delta+RLE compression and apply it to the base-centric database schema. We encode values of column `SB.Reference_Base_ID` using Delta+RLE encoding. Instead of encoding the delta values using run-length encoding, we generalize the concept of run-length encoding to support values that have a fix delta, i.e., one. This way we can compute single values within a run by adding the offset of the value to the actual run value. For example, the first run in Figure 7 has 10 as run value. If we want to decompress the third value within this run that has an index offset of 2, we sum 2 and 10 which gives us the decompressed value 12. Delta+RLE encoding can also be applied to `RB.Position`.

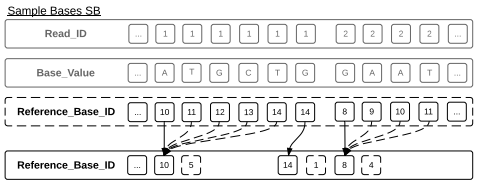


Fig. 7: Delta+RLE encoding represents runs of consecutive values as run value and length value similar to RLE encoding. This leads to an implicit string encoding for DNA sequences.

Fast data access. Delta+RLE has bad random access performance, because count values have to be summed up, in order to determine the value of a given row. Thus, we switch from using count values to prefix sums representing row ids [AMF06]. Given a row id, we can use binary search to determine the containing run. Using a caching mechanism as described in Section 4.1, subsequent sequential accesses do not require a binary search.

4.3 Base-centric schema as primary data layout

Now that we explained how to integrate genome-specific compression into a column-oriented database system, we conclude that the base-centric database schema is the more favorable data representation for genome data stored in a column-oriented database system than the straightforward sequence-centric schema. First, the base-centric database schema has similar storage requirements than the sequence-centric database schema for two reasons:

1. Column `SB.Base_Value` stores the single characters of reads consecutively, thus, the sequence of characters in memory resembles the original read sequence.
2. Using Delta+RLE encoding, we can store explicit position information highly efficient. As bases within reads are mapped to successive positions, we usually have to store one run value and one count value per read. That is similar to the storage requirements for keeping a pointer to a string per read in a sequence-centric database.

Second, the base-centric data representation encodes genome data in a database-native way that allows for direct data processing such as SNV calling (cf. Section 3). Moreover, we can leverage the base-centric schema to integrate genome-specific compression schemes. In the following, we call the database approach using the base-centric schema as primary data layout DBS_{base} to distinguish it from the straightforward approach that we now call DBS_{seq}. Note, all optimizations discussed so far can also be applied to DBS_{seq} after genome data has been converted effectively reducing the memory footprint of DBS_{seq}.

Advanced join processing required. The remaining challenge when using DBS_{base}, i.e., using the base-centric database schema as primary data layout, is the large size of table `Sample_Bases`. For example, if we store the complete genome used in Section 3,

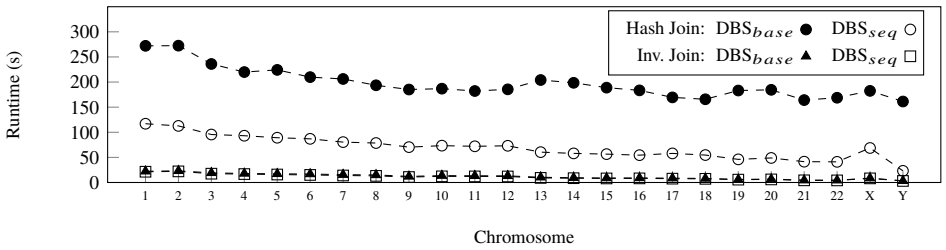


Fig. 8: Hash and invisible join runtimes on single chromosomes of a human genome using DBS$_{base}$ and DBS$_{seq}$. The invisible join is superior to the hash join and required to overcome the processing overhead introduced by the base-centric database schema due to large table sizes.

table `Sample_Bases` contains more than 14 billion rows, even if we just analyze a single chromosome. In contrast, using DBS$_{seq}$, we only convert the data required for analysis. For example, if we analyze chromosome 22, table `Sample_Bases` contains only 160 million rows. Considering the join between table `Reference_Bases` and `Sample_Bases` during SNV calling (cf. Figure 3), this difference in size becomes critical. Using a hash join, we hash table `Reference_Bases` and probe table `Sample_Bases`. Consequently, DBS$_{base}$ is slower than DBS$_{seq}$, because we always have to probe more rows. In Figure 8, we show the hash join runtimes calling single chromosomes on a human genome (cf. circled plots).

An alternative join technique is the *invisible join* proposed by Abadi et al. [AMH08]. The key idea is to apply predicates on dimension tables directly to the fact table (predicate rewriting) and to reconstruct join tuples later via positional lookups using foreign keys as indexes. A hash-based semi join between dimensions and fact table is a general strategy for predicate rewriting, but still requires to probe billions of rows of table `Sample_Bases`. In order to make the predicate rewriting more efficient, Abadi et al. introduce the so called *between-predicate rewriting*. They observed that predicates on dimension tables can often be rewritten as between-predicate on the respective foreign-key column of the fact table. In our case, we can rewrite the predicate on table `Reference_Bases` to filter for single chromosomes into a between predicate on column `SB.Reference_Base_ID` in table `Sample_Bases`, because reference bases are stored consecutively leading to consecutive primary keys. This reduces the memory footprint of our database approaches as we do not have to create intermediate hash tables [Do16]. Moreover, we only have to scan the foreign key column avoiding the effort of hash probing. Usually, the scan also has to touch every row in table `Sample_Bases`. In combination with Delta+RLE encoding, we are able to skip all rows represented by a run if the run disqualifies for the between predicate. Using the invisible join technique in combination with Delta+RLE encoding leads to large performance improvements of the join phase in DBS$_{base}$ and DBS$_{seq}$ (cf. Figure 8).

5 Base pruning

We also analyzed the general functionality of SNV callers. We came up with the conclusion that only those genome positions show a differing genotype than the reference base if at least one sample base differs from the reference base. Thus, if we know that a genome position has only matching sample bases mapped to it, we can exclude it from

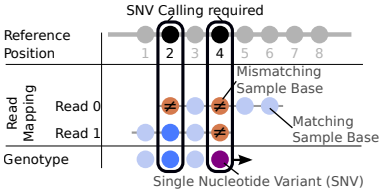


Fig. 9: Base pruning filters out genome positions where no mismatching base has been mapped which reduces the number of genome positions to process during SNV calling.

the semantics of the genotype UDF. further processing during SNV calling, i.e., applying a domain-specific filter on the data. We show the idea in Figure 9. The black boxes indicate which genome positions have to be processed. The other genome positions have no mismatching bases (circles with $\neq$ symbol) mapped to it. Thus, we can reduce the processing effort during query processing. We only have to join those sample bases that may lead to a SNV call. Finally, the join result only contains those sample-base/reference-base tuples that really need to be aggregated. A traditional optimizer is not able to apply this optimization as it has no knowledge about

5.1 Approaches

In order to compute which genome positions have no differing sample base, we have to know in advance, which sample bases are mapped to which reference base. Using the base-centric database schema for storing genome data, we have this knowledge already encoded as foreign-key relationship between tables `Sample_Bases` and `Reference_Bases`.

Straightforward approach. In a straightforward approach, we scan the `Sample_Bases` table and compare each sample base with the corresponding reference base. Thus, we collect all genome positions that have at least one differing sample base. In a second step, we scan the table `Sample_Bases` again in order to determine all sample bases that are mapped to a genome position that has at least one differing sample base. Consequently, the straightforward approach requires two table scans over table `Sample_Bases`. These can be implemented very efficient. Nevertheless, during the first scan, we look up reference bases from the table `Reference_Bases`. Within one read, these lookups are cache-efficient as a read usually maps to a consecutive region within the reference genome. Thus, we cache further accesses to subsequent reference bases. Nevertheless, this approach introduces overhead due to comparisons. Moreover, for every new read, we make a random lookup into table `Reference_Bases` as different reads do not have to map to consecutive regions leading to cache misses.

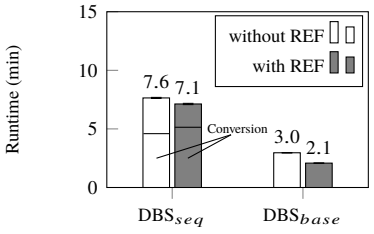


Fig. 10: Overall SNV calling runtime on a whole human genome using *base pruning*. with and without reference-based compression. Reference-based compression always improves runtime.

Indexed approach. Using reference-based compression, we can improve the straightforward *base pruning* computation. The reference-based compressed column `SB.Base_Value` already encodes which sample base is different according to the reference base. Thus, we can use it as index and extract all row ids of all differing sample bases from the bitmap of the compressed `SB.Base_Value` column. Therefore, we only have to scan the bitmap and return all row ids that are marked with one. Hence, we avoid to access table `Reference_Bases` at all.

Furthermore, we do not have to make the comparison between every sample and reference base as the result is already encoded in the bitmap. In case of high mapping quality, an optimized bitmap, e.g., a WAHBitmap, contains many zeroes, which allows for skipping all rows represented by a zero fill word. In Figure 10, we show the impact of light-weight reference-based compression on the runtime of DBS_{seq} and DBS_{base} using *base pruning*. Using DBS_{base}, we can reduce the runtime by 30%. Using DBS_{seq}, the runtime reduction is roughly 7%, because data conversion takes most of the runtime that cannot be reduced using *base pruning*. Moreover, using reference-based compression increases the conversion runtime in DBS_{seq}. Nevertheless, the overall runtime savings due to *base pruning* pay off.

5.2 Applicability to specialized analysis tools

So far, we considered *base pruning* in the context of our database approaches. We can also apply it to specialized analysis tools such as SAMTOOLS [Li09]. Nevertheless, heavy-weight compression used by state-of-the-art flat file formats limits the effectiveness:

Applicability to aggregation phase only. We can apply *base pruning* only to improve the aggregation phase of the analysis tool. The reason for this is that SAMTOOLS operates on heavy-weight compressed data. To guarantee reasonable performance, also for random lookups within genomes, samtools requires mapped reads to be sorted by their genome position before being compressed. This is essentially the grouping attribute of the aggregation. Hence, SAMTOOLS can interleave decompression and conversion process and generate a ready-to-aggregate output, a so called pileup, because consecutive reads belong to the same genome region. Consequently, before we know which bases belong to which reference base in order to apply *base pruning*, data is already ready for aggregation. Compared to DBS_{seq}, we have already computed the join result. What remains is to check which pileups contain no differing base and do not have to be aggregated saving analysis runtime.

Reference-based compression cannot be exploited. We cannot exploit reference-based compressed data as index for improving the *base pruning* computation. This is a direct consequence from the first limitation. We perform the *base pruning* computation after the data is decompressed. Thus, we already lost the advantage of exploiting reference-based compression to reduce the computational effort.

6 Evaluation

In this section, we evaluate the database-driven approaches DBS_{seq} and DBS_{base} for SNV calling with regard to runtime performance and storage consumption. First, we investigate the storage consumption and compare it with the state-of-the-art flat-file formats CRAM and BAM. We want to find out whether we can cope with the storage blowup of the base-centric database schema. Moreover, we want to investigate what data characteristics impact our compression schemes at most. Then, we examine the SNV calling runtime on three real world data sets and compare it with the state-of-the-art analysis tool SAMTOOLS 1.3 [Li09]. We are interested in the overall analysis performance and how data characteristics influence it. Moreover, we want to find out to what extent the *base pruning* technique improves analysis runtime.

Organism	*Homo sapiens*		*Hordeum vulgare*
DataSet	*1*	*2*	*3*
# Mapped Bases	13.9B	11.8B	3.9B
# Reference Bases	3.1B	249M	1.9B
∅ Coverage	4	47	2
∅ Read Length	100	250	100
Mismatch Rate %	0.3	0.8	1.4

Tab. 2: Genome data sets differ in their key characteristics, which have impact on storage consumption and processing performance.

6.1 Experimental setup

As evaluation platform, we use a machine with two Intel Xeon E5-2609 v2 with four cores @2.5 GHz and 256 GB main memory. On the software side, we use Ubuntu 14.04.3 (64 Bit) as operating system and CoGaDB as database system (cf. Section 3). To compile CoGaDB, we use gcc 4.8.4 with optimization level *-O3*. Before starting the experiments, we pre-load the database into main memory. Similar to our initial experiment in Section 3.2, we use a manually parallelized and functionally reduced version of SAMTOOLS that accesses flat files stored on a ramdisk to make the comparison fair. We report average runtimes of 30 runs. Moreover, for runtime results, we report the 95% confidence intervals.

Data Sets. For our experiments, we use three real world data sets. *DataSet 1* and *2* contain human genome data and *DataSet 3* contains barley genome data. We obtained the human genome data from the 1000 genomes project[9], which provides representative real world data sets [Th15]. *DataSet 2* contains only the mapped reads of human chromosome 1. The plant research institute IPK Gatersleben provided us with barley data.

The data sets differ in their *number of reference bases*, *coverage*, *read length* and *mismatch rate*. In Table 2, we summarize the characteristics. The *number of reference bases* indicates the upper bound for genome positions that have to be analyzed. The *coverage* indicates how many sample bases are mapped on average to a certain reference genome position. Thus, in case of SNV calling, higher coverage leads to more sample bases to aggregate per reference genome position. *Coverage* and *number of reference bases* together determine the *number of mapped bases* in a data set. The *read length* has direct impact on storage consumption. If reads are longer, less read data per sample base must be stored (cf. *DataSet 1* and *2*). The *mismatch rate* indicates how many sample bases within the data set are different from their corresponding reference base. The barley data has a higher mismatch rate than the human data sets, which can have impact on the number of reported SNVs and may impact the effectiveness of reference-based compression and base pruning.

6.2 Storage consumption

In the first experiment, we examine the storage consumption of the database approaches DBS_{seq} and DBS_{base} and the state-of-the-art flat-file formats BAM and CRAM. We have

[9] data is available at ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG00096/

Approach	Storage consumption in GB (Relative to DBS_{seq})			Main compression type
	DataSet 1	*DataSet 2*	*DataSet 3*	
DBS_{seq} (baseline)	38.0 (100%)	27.2 (100%)	12.9 (100%)	light-weight
DBS_{base}	27.2 (72%)	17.8 (65%)	9.5 (74%)	
BAM	14.6 (38%)	6.9 (25%)	3.9 (30%)	heavy-weight
CRAM	10.3 (27%)	4.9 (18%)	3.2 (25%)	
Zipped DBS_{base}	11.7 (31%)	6.1 (22%)	3.3 (26%)	

Tab. 3: Storage consumption of DBS_{seq}, DBS_{base}, BAM and CRAM. DBS_{base} outperforms DBS_{seq} on all data sets. BAM and CRAM are superior as they apply heavy-weight compression. Zipping DBS_{base} leads to a similar storage consumption as CRAM.

two main objectives: 1) We want to investigate whether we can cope with the storage blowup of the base-centric database schema and 2) We want to find out how data characteristics impact the compression ratio. We report the absolute storage requirements for storing sample genome and respective reference genome data in Table 3 including storage required for indexes to improve data access speed. In brackets, we show the relative storage requirements compared to DBS_{seq}, which serves as our baseline.

Effective reference-based compression in column-stores. The results show that DBS_{base} always requires less storage than DBS_{seq} independent of the stored data set. Due to Delta+RLE compression that effectively reduces the overhead due to explicit positional information and reference-based compression, we can decrease the storage size in a database system by 26 to 35%. Compared to BAM, DBS_{base} needs 2 to 2.5 times more storage than BAM, since we do not use heavy-weight compression. Still, reference-based compression is an essential mean to reduce the storage size of genome data. The CRAM results show that reference-based compression in combination with heavy-weight compression further reduces the storage size compared to BAM. Compressing the disk-resident data files of DBS_{base} using GZIP leads to a similar result (cf. last row of Table 3).

Data-dependent storage requirements. Table 3 reveals that the storage savings of DBS_{base} compared to DBS_{seq} depend on the data set. The reason for the differences between the three data sets is two-fold: *read length* and *mismatch rate*. In Table 4, we show the impact of these characteristics on the three columns `SB.Reference_Base_ID`, `SB.Read_ID` and `SB.Base_Value`. All other columns' sizes are independent of data set characteristics.

Impact of read length. We use RLE to compress `SB.Read_ID` and our Delta+RLE encoding to compress `SB.Reference_Base_ID`. Both encodings are sensitive to the length of runs within the data. The longer the runs, the better the compression ratio. Since we store bases of the same read consecutively (`SB.Read_ID`) and these bases usually map to successive genome positions (`SB.Reference_Base_ID`), longer reads lead to increased run length. For that reason, in *DataSet 2* with 250 bases per read on average, each column requires 0.6 bit per column per row. The other two data sets require 1.4 bit per column per row, because reads ahve an average length 100 bases.

Impact of mismatch rate. The different mismatch rates of the data sets directly impact the storage requirements of the reference-based compressed column `SB.Base_Value`. Fewer mismatches lead to fewer values to be stored in the exception list. Moreover, the bitmap contains more zeroes that can be leveraged by a WAHBitmap. Therefore, *DataSet 1* requires

DataSet		*1*	*2*	*3*
# Mapped Bases		13.9B	11.8B	3.9B
∅ Read Length		100	250	100
Mismatch Rate %		0.3	0.8	1.4
Bits per row	`SB.Reference_Base_ID`	1.4	0.6	1.4
	`SB.Read_ID`	1.4	0.6	1.4
	`SB.Base_Value`	0.6	0.8	1.3
	Sum of bits	3.4	2.0	4.1

Tab. 4: Influence of data characteristics on storage consumption using DBS_{base}. The longer the reads, the smaller is the overhead of foreign key columns `SB.Reference_Base_ID` and `SB.Read_ID`.

only 0.6 bits on average per mapped base. The other data sets require more bits per mapped base in concordance with their mismatch rate. Among the three columns, the overall storage consumption is dominated by the read length. The overall required number of bits per row in the three columns corresponds to the observed storage savings of DBS_{base} comapred to DBS_{seq} (cf. Table 3).

6.3 SNV calling runtime

In the second experiment, we examine the SNV calling runtime of DBS_{base}, DBS_{seq} and SAMTOOLS 1.3 with and without base pruning. Note, we use the same probabilistic error-model in our database approaches as SAMTOOLS. We do not consider post-processing validations that can be applied to the results of all approaches. We show the runtime results on the three different data sets from the experiment before in Figure 11. The hatched bars indicate runtimes with *base pruning*. First, we consider the runtime of the single approaches *without base pruning* and investigate the impact of data characteristics. Then, we examine the impact of *base pruning*.

A base-centric database schema pays off. As expected, DBS_{base} always outperforms DBS_{seq} if we do not apply base pruning, because we do not have to convert data on-the-fly. Although DBS_{base} has to process all sample bases, in particular during the join phase, we can reduce the overhead effectively using the invisible join technique (cf. Section 4.3). Moreover, the experiment reveals that DBS_{base} is competitive in terms of runtime compared to SAMTOOLS due to the use of advanced processing techniques. Thus, the required effort to compress base-centric genome data pays off.

Impact of data characteristics. Considering the results *without base pruning*, we find that the runtime depends on the number of genome positions and mapped bases to process.

Number of genome positions. For example, *DataSet 1* and *DataSet 2* contain roughly the same amount of mapped bases to process, but *DataSet 1* contains data for the complete genome, i.e. 3.1 billion genome positions. In *DataSet 2* all mapped bases only belong to the 249 million genome positions of chromosome 1. We expected a correlation with the overall analysis runtime, because computing more genome positions requires managing

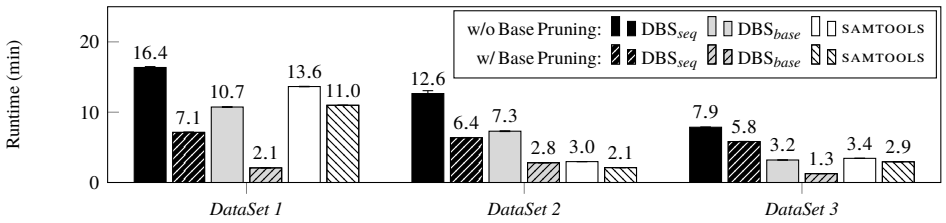


Fig. 11: SNV calling on three different real-world data sets using DBS$_{base}$, DBS$_{seq}$ and SAMTOOLS with and without *base pruning*. DBS$_{seq}$ is always slower due to conversion overhead. DBS$_{base}$ benefits most from base pruning.

more aggregation groups. Nevertheless, the large runtime differences of SAMTOOLS between *DataSet 1* and *2* are unexpected. The database approaches roughly require 30% less runtime on *DataSet 2* compared to *DataSet 1*, because less groups have to be initialized and computed. In contrast, SAMTOOLS saves 80%. An indepth analysis of SAMTOOLS revealed that SAMTOOLS has large overhead for writing analysis results per genome position via strings. Thus, analyzing many genome positions (cf. Dataset 1) increases its runtime.

Number of mapped bases. The runtimes on all data sets reveal that the database approaches are more affected by the number of mapped bases to process. Considering *DataSet 1*, the runtime of DBS$_{base}$ and SAMTOOLS are nearly equal, but for *DataSet 2*, using DBS$_{base}$ increases the runtime by factor 2.5 compared to SAMTOOLS. The reason for this difference is that the database approaches suffer from sorting or synchronization overhead during aggregation processing. SAMTOOLS requires presorted data allowing for filterings by genome positions and manual parallelization. The presorting also ensures that different threads operate on distinct genome regions. We can emulate this behavior by using a sort-based aggragtion processing, which obviously introduces overhead during runtime. Another strategy is a hash-based aggregation. Certainly, this strategy requires locking mechanisms, because the work per genome region is distributed between different threads. Consequently, the database aproaches suffer from high coverage data.

The tradeoff of both data characteristics can be seen when processing *DataSet 3*. Again the runtimes of DBS$_{base}$ and SAMTOOLS are competitive. *DataSet 3* contains less genome positions favoring SAMTOOLS, but also less mapped bases to process favoring DBS$_{base}$.

Impact of base pruning.

Using *base pruning*, we aim to restrict the costly processing to those genome positions that might show a variant (cf. Section 5). Using one of the database approaches, we can make benefit of this reduction during the join and aggregation phase. Using SAMTOOLS, we still have to decompress, convert and inspect all genome positions before taking advantage of the *base pruning* technique (cf. Section 5.2). Consequently, the database approaches benefit most from using *base pruning*. In Figure 11, we show the runtime results of all three approaches *with base pruning* indicated with hatched bars.

DBS$_{base}$ outperforms SAMTOOLS on *DataSet 1* and *3*. Even DBS$_{seq}$ is faster than SAMTOOLS on *DataSet 1*. On *DataSet 2*, SAMTOOLS is still faster due to less genome positions to process even without base pruning. Overall DBS$_{base}$ benefits most from *base pruning*, because we reduce the number of genome positions to process during all processing steps.

6.4 Discussion

Our evaluation shows that a base-centric data representation outperforms a straightforward database approach regarding storage consumption. The concrete storage savings depend on the read length and number of mismatching bases. Nevertheless, heavy-weight compression impacts storage size more, even on a sequence-centric data representation. Thus, BAM, which applies only heavy-weight compression, requires less storage than DBS_{base}. CRAM that additionally applies reference-based compression can compress data even better. Our experiments show that a simple compression of database files using GZIP leads to similar results than CRAM. Thus, integrating heavy-weight compression into our database approach would be beneficial especially to keep cold data without overhead on secondary storage.

The second advantage of a base-centric data representation for genome data is the improved analysis performance compared to a straightforward approach. We are able to detect SNVs as efficiently as SAMTOOLS. To achieve this performance, we rely on advanced processing techniques such as invisible join. In combination with *base pruning*, we can further improve analysis runtime. Overall, our proposed techniques and approaches benefit from longer reads and smaller mismatch rate. Thus, database systems using our techniques will benefit from improvements in DNA sequencing that will generate longer and more accurate reads [Li12] leading to less mismatches due to mapping errors.

7 Related work

In the following, we categorize and discuss approaches that use database systems and technology to efficiently store, manage and analyze genome data.

Data warehouse approaches. One of the first approaches to manage and integrate genome data in a database system is AceDB [ST99] using an object database. Several scientists proposed more advanced data warehouse solutions for managing and analyzing genome data and related data from other data sources [Le06; Sh05; Tö08]. The main focus of these solutions is the integration of different heterogeneous data sources to allow for integrated analyses. These approaches do not consider storage size, analysis efficiency or incorporate genome analysis tasks such as SNV calling. Instead they integrate such data. Our proposed approaches complement these data warehouse solutions by integrating SNV calling into a DBS. Moreover, we propose compression schemes that enable a data warehouse to store raw data and compute analysis results on-the-fly.

Integrated data analysis. Besides classical data warehouse solutions, approaches exist that integrate genome analysis functionality into a data management solution. For example, Ceri et al. present a data-management approach that allows for storing and querying genome-position specific data using a simple data model called Genomic Data Model [Ce16]. Furthermore, they propose the GenoMetric Query Language that provides algebraic operations similar to SQL and domain-specific analysis functionality. In contrast, our approach aims at using existing database technology to support genome analysis tasks. *bdbms* proposed by Eltabakh et al. also extends an existing database system with biological functionality [EOA07] such as annotations and provenance tracking. Moreover, it provides pattern matching functionality for compressed sequence data. Our work complements *bdbms* by providing genome-specific compression and analysis functionality.

Our work is mainly related to the work by Röhm and Blakeley [RB09]. They propose an approach to integrate genotyping, the pre-computation step of SNV calling, into a database system. They use a disk-based database system and enable users to operate on the original flat-files. Nevertheless, they report unsatisfactory analysis performance due to the use of multiple user-defined functions that are hard to parallelize. Moreover, within their approach for genotyping, they follow a straightforward flat-file approach introducing additional conversion overhead (cf. Section 3). Our DBS$_{base}$ approach avoids this overhead using a special encoding of genome data requiring only one user-defined function for analysis.

Moreover, several other approaches exist that explicitly use main-memory database systems to integrate genome analysis tasks. The approach by Fähnrich et al. uses a two-phase map-reduce approach to convert reads and perform SNV calling [FSP15]. The approach by Cijvat et al. uses a special user-defined function of MonetDB to convert DNA sequences [Ci15]. As this operation is expensive, they cache the result for further analysis. Thus, our proposed technique to efficiently encode converted genome data complement both approaches.

Reference-based analysis techniques. Currently, we are aware of one variant calling approach called CAGe that leverages the similarity between reference genome and sample genome to reduce the analysis runtime of variant detection [Bl14]. The approach classifies genome regions regarding their analysis complexity incorporating information about similarity. Regions with high similarities have a low complexity and are analyzed using fast variant calling approaches. On the other hand, regions with many differences are complex and more sophisticated approaches are applied. Our base-pruning approach can improve the overall analysis runtime as it reduces the runtime to analyze low complexity regions.

Another approach that leverages the similarity between reference and sample genome is RCSI proposed by Wandelt et al. [Wa13]. This approach aims at similarity search on referentially compressed genomes. The idea is to first search on the reference sequence finding matching segments that may contain errors. In a second step, the compressed sample genomes are searched at the corresponding segments to generate the final result. Our approach to integrate reference-based compression provides a basis to integrate this technique into a relational database system. Moreover, we can use the optimized database processing engine to look up sample genome segments of interest fast.

8 Conclusion

In this paper, we showed that a base-centric data representation is required to integrate genome-specific compression schemes such as reference-based compression into a database system. Based on this, we proposed a filtering technique called *base pruning* leveraging reference-based compression as index. Using our database-native approach improves the overall runtime up to a factor of five compared to specialized analysis tools. The concrete performance gains depend mainly on coverage and mismatch rate of the analyzed data set.

Overall, our techniques enable scientists and researchers to perform SNV calling within a database on-the-fly, instead of precomputing results. In our experiments, we used a probabilistic calling approach based on the error-model routine of SAMTOOLS. By simply choosing a different aggregation function, we can apply an alternative variant calling approach if necessary [Ni11]. Especially on small genome regions, an interactive and

declarative analysis becomes possible. The raw data and analysis results are delivered by a single database system improving traceability of results. Additionally, we will benefit from future improvements of database systems due to our database-native application design. The code used in this paper is available at http://cogadb.dfki.de/download/.

Acknowledgements

The work has received funding from the German Research Foundation (DFG), Collaborative Research Center SFB 876, project C5, from the European Union's Horizon2020 Research & Innovation Program under grant agreement 671500 (project SAGE), and by the German Ministry for Education and Research as Berlin Big Data Center BBDC (01IS14013A).

References

[AMF06] Abadi, D. et al.: Integrating compression and execution in column-oriented database systems. In: SIGMOD. pp. 671–682, 2006.

[AMH08] Abadi, D. et al.: Column-Stores vs. Row-Stores: How different are they really? In: SIGMOD. pp. 967–980, 2008.

[BFT16] Breß, S. et al.: Robust query processing in co-processor-accelerated databases. In: SIGMOD. pp. 1891–1906, 2016.

[Bh04] Bhagwat, D. et al.: An annotation management system for relational databases. In: VLDB. pp. 900–911, 2004.

[Bl14] Bloniarz, A. et al.: Changepoint analysis for efficient variant calling. In: RECOMB. pp. 20–34, 2014.

[Br13] Bromberg, Y.: Building a genome analysis pipeline to predict disease risk and prevent disease. J. Mol. Biol. 425/21, pp. 3993–4005, 2013.

[Ce16] Ceri, S. et al.: Data management for next generation genomic computing. In: EDBT. pp. 485–490, 2016.

[Ci15] Cijvat, R. et al.: Genome sequence analysis with MonetDB - A case study on Ebola virus diversity. Datenbank-Spektrum 6/17, pp. 185–191, 2015.

[CR15] CRAM Format Spec. Working Group: CRAM Format Specification, 2015.

[DBS14] Dorok, S. et al.: Toward efficient variant calling inside main-memory database systems. In: BIOKDD-DEXA. pp. 41–45, 2014.

[De11] DePristo, M. et al.: A framework for variation discovery and genotyping using next-generation DNA sequencing data. Nat. Genet. 43/5, pp. 491–498, 2011.

[Do16] Dorok, S.: Memory efficient processing of DNA sequences in relational main-memory database systems. In: GvDB. pp. 39–43, 2016.

[EOA07] Eltabakh, M. et al.: bdbms - A database management system for biological data. In: CIDR. pp. 196–206, 2007.

[FSP15] Fähnrich, C. et al.: Facing the genome data deluge: Efficiently identifying genetic variants with in-memory database technology. In: SAC. pp. 18–25, 2015.

[Hs11] Hsi-Yang Fritz, M. et al.: Efficient storage of high throughput DNA sequencing data using reference-based compression. Genome Res. 21/5, pp. 734–740, 2011.

[Ku07] Kuenne, C. et al.: Using Data Warehouse Technology in Crop Plant Bioinformatics. J. Integr. Bioinform. 4/1, 2007.

[Le06] Lee, T. J. et al.: BioWarehouse: A bioinformatics database warehouse toolkit. BMC Bioinformatics 7/1, pp. 170+, 2006.

[LH10] Li, H. et al.: A survey of sequence alignment algorithms for next-generation sequencing. Brief. Bioinform. 11/5, pp. 473–483, 2010.

[Li09] Li, H. et al.: The Sequence Alignment/Map format and SAMtools. Bioinformatics 25/16, pp. 2078–2079, 2009.

[Li12] Liu, L. et al.: Comparison of next-generation sequencing systems. J. Biomed. Biotechnol. 2012/, pp. 1–11, 2012.

[Ma10] Mardis, E. R.: The $1,000 genome, the $100,000 analysis? Genome Med. 2/11, pp. 1–3, 2010.

[Ma13] Mavaddat, N. et al.: Cancer risks for BRCA1 and BRCA2 mutation carriers: Results from prospective analysis of EMBRACE. J. Natl. Cancer Inst. 105/11, pp. 812–22, 2013.

[Ni11] Nielsen, R. et al.: Genotype and SNP calling from next-generation sequencing data. Nat. Rev. Genet. 12/6, pp. 443–51, 2011.

[Qu12] Quail, M. et al.: A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers. BMC Genomics 13/1, pp. 341+, 2012.

[RB09] Röhm, U. et al.: Data management for high-throughput genomics. In: CIDR. 2009.

[Sa13] Sandve, G. K. et al.: Ten simple rules for reproducible computational research. PLoS Comput. Biol. 9/10, 2013.

[SA15] SAM/BAM Format Spec. Working Group: SAM Format Specification, 2015.

[Sh05] Shah, S. P. et al.: Atlas - A data warehouse for integrative bioinformatics. BMC Bioinformatics 6/1, pp. 34+, 2005.

[ST99] Stein, L. D. et al.: AceDB: A genome database management system. Comput. Sci. Eng. 1/3, pp. 44–52, 1999.

[Th15] The 1000 Genomes Project Consortium: A global reference for human genetic variation. Nature 526/7571, pp. 68–74, 2015.

[Tö08] Töpel, T. et al.: BioDWH: A data warehouse kit for life science data integration. J. Integr. Bioinform. 5/2, 2008.

[Wa13] Wandelt, S. et al.: RCSI: Scalable similarity search in thousand(s) of genomes. PVLDB 6/13, pp. 1534–1545, 2013.

[WOS06] Wu, K. et al.: Optimizing bitmap indices with efficient compression. ACM Trans. Database Syst. 31/1, pp. 1–38, Mar. 2006.

Index Structures

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 445

Quadtree-based Resource Description Techniques for Spatial Data in Distributed Databases

Stefan Kufer[1], Andreas Henrich[1]

Abstract: In the social media age, content creation and distribution of all sorts of digital media is of growing importance. The variety of media types, devices, and user groups results in the generated media items being stored in a very heterogeneous landscape which can be seen as a large distributed database consisting of personal computers, mobile devices, or web servers of social media sites, demanding for an appropriate overarching search system. Resource selection based on compact resource descriptions allows to efficiently determine resources maintaining relevant media items. Among other aspects, geospatial footprints have to be addressed for an effective media search.

We propose and evaluate quadtree-based techniques to summarize collections of geo-referenced media items in a distributed geographic IR context. Based on these summaries, resource selection decisions are made when searching for media items close to a given geographic location. The quadtree-based approaches are evaluated against previous work. The quadtree-based summaries prove to be very flexible and competetive.

Keywords: Geographic Information Retrieval, Distributed Information Retrieval, Resource Selection, Summarization, Spatial Data

1 Introduction

In distributed retrieval settings, two principal approaches can be distinguished. In the first setting, the data is assigned to the distributed resources (i.e. to the nodes in the network) based on its attribute values. Considering spatial attributes this could mean that each resource maintains data items spatially located in a certain area. Of course, this setting allows for an efficient processing for spatial nearest neighbor queries. However, the distribution of the data to the resources can be based only on one attribute and it might not be appropriate at all in certain application scenarios. This brings us to the second scenario of distributed retrieval, where the data is maintained in different resources simply because of its origin or other circumstances. An example could be geo-tagged media items—images for example—which are maintained in the personal media archives of people or organizations participating in a network.

Especially in the second scenario, effective and efficient resource description and selection techniques are important for distributed retrieval. Obviously, the media items usually have different attributes: low level image features, tags or short textual descriptions, timestamps, and a geographic position where the image was taken. Queries for media items can address all these aspects. Consequently, summaries (resource descriptions) for all these aspects should be distributed in the network (e.g. via rumor spreading [Cu03]) to allow for a targeted

[1] Otto-Friedrich-Universität Bamberg, Lehrstuhl für Medieninformatik, An der Weberei 5, D-96047 Bamberg,
(stefan.kufer|andreas.henrich)@uni-bamberg.de

selection of the resources relevant to a certain query. Because of its two-dimensional nature and its descriptive power the spatial information can play an important role here. On the other hand, simple approaches for summarization, like bounding boxes, obviously are not appropriate. Assume for example a person usually taking pictures in Upper Franconia who in addition has some images in her collection taken at her last trip to the Olympic Games in Brazil. In this case, a bounding box would not be appropriate to describe these two spots.

In the paper at hand we present and evaluate quadtree-based approaches in comparison to previously published approaches based for example on multiple rectangles or Voronoi-like space partitionings. It turns out that quadtree-based approaches are well-suited especially when extremely compact resource descriptions are desired.

2 Problem Description and Preliminaries

In a multi-database model, the existence of multiple databases is explicitely modelled. It is scalable to large numbers, but introduces some additional complexity compared to the single-database model [Ca00, 127f]:

Resource Description Problem A brief description of the contents of a database or resource is required, facilitating the identification of resources providing relevant information.

Resource Selection Problem A subset of resources that (most likely) contain relevant information is selected (based on the resource descriptions) to contact while querying.

Result Merging The returned results of the selected subset of resources to be contacted have to be integrated and merged into an overall result.

For our distributed search scenario, we assume that every resource in the network maintains a set of geotagged media items, meaning each media item is enhanced with a single pair of latitude/longitude (lat/lon) coordinates. The geo-coordinates will be treated as plate-carrée-projected data points, resulting in lat/lon coordinates corresponding to x/y coordinates in a two-dimensional Cartesian coordinate system (with lon being x). Since the data points thus are encoded in a two-dimensional Euclidean space, we use the Euclidean distance for distance calculations. Note that former work [BH12] has investigated distance measures better suited for distance calculations on the earth's surface, namely the Vincenty distance and the Haversine distance, but did not result in relevant differences compared to the computationally less complex Euclidean distance.

Though the data is bounded (-90 to 90 in lat $= y$ and -180 to 180 in lon $= x$), the Date Line is taken into account for *all* distance calculations (point-to-point, point-to-rectangle, etc.), considering the data stems from lat/lon coordinates on the earth. Consequently, the distance between for example two points $d_1 = (179; 0)$ and $d_2 = (-179; 0)$ is 2 rather than 358. For this work, we take a static perspective on the database, meaning the number of data points per resource will not grow and shrink. In a 'real-world' scenario, the resource description techniques of course are qualified for dynamic data—by recalculating the respective resource description of a resource whose set of data points alters.

3 Resource Descriptions for Geospatial Data

For the Resource Description Task, the objective is to encode sets of two-dimensional data points effectively (accurate description) and efficiently (compact storage). Furthermore, a fast execution of the geometric search operations is also desirable. Previous work ([KBH12], [KBH13], [KH14], and on a more abstract level [BHK16]) has investigated several techniques distinguishable into three categories, which briefly are recapitulated in section 3.1. After clarifying some prerequisites for the use of quadtrees in our scenario (section 3.2), the quadtree-based techniques newly developed for this work are presented in section 3.3.

3.1 Basic and State of the Art Resource Description Techniques

Generally, in the context of Point Access Methods (PAMs), it is often distinguished between techniques which *organize the data* and techniques which *organize the embedding space* [Sa05, p. 2f]. We adapt this categorization and differentiate *Geometric Approaches* which organize the data and *Space Partitioning Approaches* which organize the embedding space. Furthermore, the properties of two arbitrary techniques can be combined into a new technique, which we classify as *Hybrid Approaches*.

Geometric Approaches Techniques of this category are using one or several bounding volumes to delimit a set of data points and thus *organize the data*. The two properties to be decided are the shape of the bounding volumes and the quantity of bounding volumes used.

For the shape, axis-aligned rectangles are most common: they are easy to compute, have a small memory footprint, and facilitate fast intersection and distance tests. Using a single shape for describing the data often is not an adequate solution, since the resource's data points can be accumulated at widely separated locations with a lot of 'dead space' inside the bounding volume. Thus, a resource's point cloud has to be divided into disjoint groups of spatially adjacent data points which then each are approximated by a bounding volume. Generally, clustering algorithms and algorithms for object decomposition of complex spatial objects are applicable for that. In the following, we will present the MBR as a basic single shape technique and the RecMAR$_{k,sl}$ as a technique utilizing several boxes.

MBR The Minimum Bounding Rectangle (MBR), also known as bounding box or envelope [Ca05, p. 1], is a rectangle whose sides are parallel to the coordinate axes of the space [Sa05, p. 195]. It is the smallest rectangle bounding a set of spatial features (the resource's data points in our case) and can be specified by the lower left corner and the upper right corner. The MBR is one of the most basic spatial data representations and will serve as a comparative scale for the other approaches. The MBR might cross the Date Line.[2]

For the resource description, the coordinates of the lower left corner and the upper right corner are captured in single-precision floating-point format, which occupies 32 bits for each value. The four necessary values for encoding the two-dimensional rectangle are linearly stored in a bit vector (see Figure 3 for an overview on the encoding schemes).

[2] Date Line crossing boxes can also occur for other Geometric Approaches or Hybrid Approaches utilizing bounding volumes as a foundation, since the computation of geometric shapes does not require bounded data spaces; thus the Date Line is not considered as a boundary for the x dimension for these techniques. This also applies for the polygons of the Voronoi diagrams' cells.

RecMAR$_{k,sl}$ The Recursive Minimum Area Rectangles (RecMAR$_{k,sl}$) approach is based on an algorithm developed by Becker et al. [Be92] for approximating a set of axes-parallel rectangles by two axes-parallel MARs, which—among all the pairs (s, t) of data point enclosing rectangles inside the overall MBR m—finds the pair for which the sum of the areas of s and t is minimal. The algorithm separately looks for different types of solutions which are distinguished depending on (a) the adjacency of the sides of s and the boundary of m and (b) the overlap between s and t. The best one of these solutions is selected. We adjusted the algorithm for the use with point data.

Furthermore, the algorithm can be applied recursively to compute up to k MARs, since it decomposes the arbitrary MBR m into the two MARs s and t (which contain all of m's data points). In the set $M = \{m_0, \ldots, m_n; n < k\}$ of already computed rectangles, the rectangle m_i whose area is largest is taken from M. m_i then is decomposed into the two MARs $m_{i.s}$ and $m_{i.t}$, which afterwards replace m_i in M. The recursion stops if either the maximum of k rectangles has been reached or the distance between the center of a rectangle and each of its associated data points is smaller than a threshold value sl for all rectangles. Conceptually, RecMAR$_{k,sl}$ is closely related to split strategies known from R-tree based approaches.

Space Partitioning Approaches The principle of methods describing the embedding space is to decompose the space into disjoint subspaces which then are exploited to identify regions (not) containing data points. Here, hierarchical and non-hierarchical methods can be distinguished. Hierarchical decomposition is based on the principle of recursive decomposition of the space [Sa05, p. XIX] that is represented by a corresponding tree structure which can be binary (such as for the kd-tree family) or multiway (such as the quadtree family). Non-hierarchical decomposition divides the space without the utilization of an underlying tree structure (for example grid-based methods).

Generally, the resulting subspaces are distinguished into regions containing data points (*occupied cells*) and regions not containing data points (*non-occupied cells*). Binary or quantitative information about cell occupancy can be associated with each cell. Another important distinction is whether the space decomposition is *the same for all resources* or if it may be *individual* for each resource. This depends on the question if the information necessary to (re)construct the subspace boundaries can efficiently be represented. If so, resource-individual space decomposition is applicable.

UFS$_{n,cc}$ The Ultra Fine-grained Summaries are using a Voronoi diagram to partition the underlying space. Generally, the Voronoi diagram of a given set $S = \{s_1, ..., s_n\}$ of n sites in R^d partitions the space of R^d into n regions, one per site. Each region includes all points in R^d with a common closest site in the given set S according to a distance metric D (Euclidean distance in our case) [Sa05, p. 346]. The Voronoi diagram is a non-hierarchical space decomposition which is not suited for resource-individual space decomposition, since either the coordinates of the sites or the corresponding polygons of the Voronoi cells are required to determine the locations of the subspaces. This information cannot be encoded efficiently in a resource's description given several hundred or thousand sites. Hence, the UFS$_{n,cc}$ approach utilizes a 'global' Voronoi diagram of a given set $S = \{s_1, ..., s_n\}$ which is the same for all resources. The distinctive quality of the space decomposition is dependent on the selection of appropriate sites [HB10], for which we randomly chose data points right

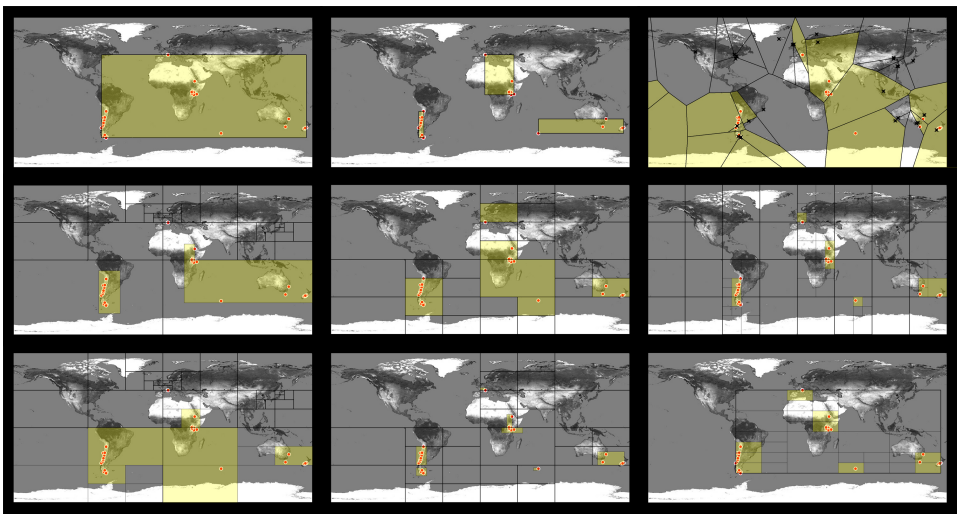


Fig. 1: Visualization of the techniques presented in this paper for the example resource with 2,186 data points. Left to right, top to bottom: MBR, RecMAR$_{3,0.1}$, UFS$_{32,x}$, KDMBR$^{3}_{32}$, QT$_{32,1.0}$, GridQT$^{8,1.0}_{4}$, KDQT$^{8,1.0}_{32}$, QTMBR$^{3}_{32,1.0}$, MBRQT$^{32,1.0}$. The red dots denote data points of the resources, the yellow surfaces indicate the regions containing data points which are described by the summaries.

out of the data collection. Not all sites have to be considered when ranking the resources. The parameter *cc* controls how many sites are taken into account (also see section 4).

Binary information about cell occupancy is captured. The resource description is represented by a bit vector with consecutive occupancy information for all n subspaces.

Hybrid Approaches Hybrid Approaches combine properties of two description methods for more effective results. The combination of one method each from the two aforementioned categories often achieves particular synergy, though it is also possible to combine two methods from the same class. Regardless of the specific methods combined, a two-step approach is used for all hybrid methods: In the first step, method `A` is used to build the foundation of the description. In the second step, method `B` is added as a refinement for the foundation.

KDMBR$^{b}_{n}$ This technique is a hybrid approach combining the organization of the embedding space known from kd-trees as a foundation with the utilization of an MBR as a bounding volume for the refinement of each occupied cell of the kd-structure.

A kd-tree is a binary search tree where the underlying space recursively is partitioned on the basis of the value of just one attribute at each level [Sa05, p. 49]. Generally, in a kd-tree, data points are organized in buckets which are split when an overflow occurs. With regard to the resource descriptions, we will examine a global kd space partition (i.e. it is the same for all resources). Thus, for the KDMBR$^{b}_{n}$ method, training data is used to learn the global space decomposition. Similarly to UFS$_{n,cc}$, the quality of the space decomposition is dependent on the selection of appropriate training data, for which we randomly chose data points right

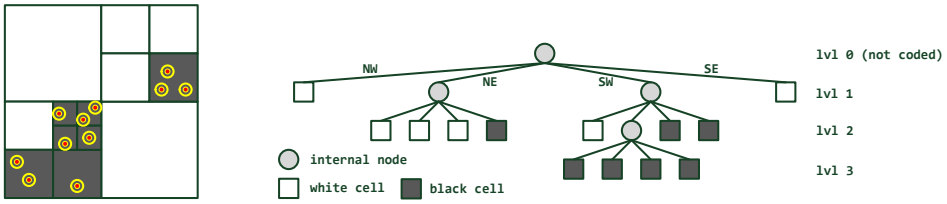


Fig. 2: Exemplary data point set depicted by a quadtree (left) and the matching tree structure (right).

out of the data collection, again. Since it is a bucket-based process, the universe initially consists of only one bucket. When a bucket-overflow occurs, the bucket is split into two. The split dimension is cyclically altered and the respective dimension is split into halves. After a split, the data insertion procedure is continued. The whole process continues until an amount of n buckets has been reached.

As a refinement, for each occupied cell of the kd space partition, the set of data points located in this cell additionally is bounded by a cell-interior MBR which is quantized for storage efficiency. This strategy has originally been introduced with the buddy-tree [SK90]. Specifically, the rectangle representing a cell of the kd-structure is called the *potential* data region. The MBR containing all of a region's data points is called the *actual* data region. It is conservatively approximated by a multidimensional interval which is quantized into a grid of $2^{b \cdot d}$ cells, which is invoked onto the potential data region and thus exploits the potential data region's presence. These *coded actual* data regions are slightly bigger than the actual data regions but save storage space in the description—both dependent on the parameter b, which specifies the amount of bits spent per bound in each dimension (d is the dimensionality of the space, thus 2 in our case).

In the resource description, binary information about the occupancy of the cells of the kd-structure is captured consecutively in a bit vector. If a cell is occupied with at least one data point, the $4 \cdot b$ bits for the corresponding quantized MBR immediately follow.

3.2 Quadtree-based Resource Description Techniques

The quadtree is a generic name for all kinds of trees that are built by recursive division of space [Oo99, p. 391]. Although the term quadtree usually refers to the two-dimensional variant (which divides the space into four quadrants, typically referred to as NW, NE, SW, and SE quadrant), the basic idea applies to an arbitrary number of dimensions d. Basically, there are two types of quadtrees [Sa05, p. 28]: The point quadtree, where the subdivision lines are based on the values of the data points; in contrast, the trie-based type forms a regular decomposition of the embedding space from which the data points are drawn into congruent regions of equal size. In the following, we are interested in the trie-based type (see Figure 2).

For a concrete utilization of quadtrees, we require some consideration concerning the decomposition of the space into subspaces and the quadtree encoding. We will present the quadtree as a solitary technique plus several methods which utilize quadtree structures in Hybrid Approaches.

Rules for the Space Decomposition In general, we are interested in binary quadtrees: the single quadtree regions (also called cells or leaves of the quadtree) are labelled black (white) if they contain at least one data object (no data object). For binary quadtrees and *region data*, the space decomposition is applied until each resulting cell is homogeneous, that is the cells are fully black or white [Sa05, p. 211]. Since we are concerned with *point data*, we need to adapt and must define a sensible stopping criterion for the decomposition. For this work, the stopping criterion is primarily storage-oriented and is applied when a certain amount of cells c is reached. Initially starting with one cell for the whole data space, the biggest black cell is equally divided into four (assuming this is the area which can be further delimited most). If there are several equally sized black cells, a random choice is made. After the divide, the resulting sibling nodes are respectively labelled black or white. The space decomposition continues until an amount of c cells has been reached, but ends prematurely if the area of all black cells falls below a certain threshold area a. This serves as a second, selectivity-oriented stopping criterion.

Quadtree Encoding For a memory efficient representation, the linear storage of quadtrees has been proposed, where a list of values stores the hierarchical tree structure [MRJ02, p. 516]. Encoding schemes can be distinguished whether the values are encoded in depth-first-order (df-order) or breadth-first-order (bf-order) and if only black nodes (= black leaves of the quadtree structure) or if all leaf nodes and internal nodes of the quadtree are encoded. In this work, we evaluate two different encoding schemes: the linear quadtree (LQ) [Ga82] and the CBLQ code [Li97].

The linear quadtree (**LQ code**) is a df-order, only black nodes encoding. A black node is identified by a unique key derived from its ordered list of ancestors. The key of successive digits represents the quadrant subdivision from which the black node originates according to a df-traversing (digits: 0 for NW, 1 for NE, 2 for SW, 3 for SE). Keys consist of up to l digits, where l is the number of levels or depth of the quadtree. If a black node is at level i ($i < l$), then only i digits are obtained, followed by a marker X (this is an adaption versus the LQ code described in [Ga82], where $l - i$ markers would follow). A condensation of the quadtree can be applied: if four cells have the same code except for the last digit, the last digit is replaced by marker X and the four cells are pooled into one (for example, 310-311-312-313 becomes 31X, dashes inserted for readabilty). Surplus markers X are stripped off of the LQ code. The quadtree in Figure 2 would be represented by 13X-210-211-212-213-22X-23X or 13-21-22-23 after condensation. Since there are five different literals to encode (0-3 and X), 3 bits are needed for the binary representation of a literal.

The **CBLQ code** is a bf-order, all leaves and internal nodes encoding. The authors claim that on average, the required storage space is only 22.2% of the LQ code. In the CBLQ code, each black (white) leaf is coded by 1 (0). The numeral 2 encodes an internal node if at least one of its descendants is an internal node. If all descendants are leaves, the node is encoded by 3. The root node is not encoded. A condensation of the quadtree can be applied, too, that is four black siblings are merged into their father node. Hence, the quadtree in Figure 2 would be represented by 0320-0001-0311-1111 or 0330-0001-0111 after condensation (dashes only included for readability). Since there are four different literals (0-3), 2 bits are needed to encode one literal.

3.3 Novel Quadtree-based Techniques

Based on these fundamentals, we will now describe the quadtree-based resource descriptions newly introduced in this paper.

$\mathbf{QT}_{c,a}$ The 'solitary' quadtree ($\mathrm{QT}_{c,a}$) is a method conducting hierarchical, resource-individual space decomposition. The quadtree for describing the resource's regions which contain data points is built according to the general rules just discussed, that is dependent on the parameters c (specifying the maximum number of quadtree regions) and a (specifying the lower-bound threshold area for quadtree regions to not be partitioned any further). Condensation is applied for the solitary quadtree.

For the resource descriptions, the quadtree information is captured in a bit vector for both encoding schemes (LQ code and CBLQ code).

The LQ scheme requires additional metadata besides the raw quadtree data to facilitate an unambiguous decoding: (1) The number of levels (#lvl) or depth of the specific quadtree must be captured, since in the LQ scheme, there is no marker to tag the end of a black node's encoding when it is at the deepest level of the quadtree. The theoretical maximum level of a quadtree with c cells is $\lfloor c/3 \rfloor = l$. Therefore, $\lceil log_2\, l \rceil$ bits are required to unambiguously encode the depth of a quadtree. (2) The number of occupied quadtree regions (#oqr) is captured right after the level of the quadtree. This information is not stringently required for the solitary quadtree, but for the hybrid methods utilizing a quadtree. See the $\mathrm{GridQT}_r^{c,a}$ paragraph for further explanation. Nevertheless, it is captured for the solitary quadtree, too, since it only requires $\lceil log_2\, c \rceil$ bits and allows for a unified encoding and decoding procedure. Afterwards, the raw quadtree data is captured in the bit vector for the LQ scheme.

For the CBLQ scheme, only the raw quadtree information (bf-order, all leafs and internal nodes encoding) is captured.

$\mathbf{GridQT}_r^{c,a}$ The $\mathrm{GridQT}_r^{c,a}$ method is a hybrid technique combining a uniform grid as a foundation with a grid-cell-interior quadtree as a refinement for occupied cells.

A (uniform) grid is a non-hierarchical structure [Sa05, p. 10]: The space is subdivided into equal sized grid cells. It is imposed onto the universe with r rows and $2 \cdot r$ columns. For each grid cell occupied with at least one data point, a cell-interior quadtree is built, dependent on the parameters c and a. For each quadtree, condensation is applied if possible.

In the bit vector for the resource description, binary information about the cell occupancy is consecutively captured. For an occupied cell, the quadtree information immediately follows. Beside the raw quadtree information, some additional information must be captured to unambiguously determine the tail of the quadtree data and the continuation of the grid cell occupancy information:

For the LQ code, the level/depth and the number of occupied quadtree regions (#oqr) of the cell-interior quadtrees must be captured. Again, the level or depth of the quadtree is encoded with $\lceil log_2\, l \rceil$ bits. The maximum number of occupied leaf nodes in a quadtree where condensation is applied is $c - 1$, (if all quadtree regions are occupied, the quadtree would

be condensed into its root node). Therefore, it requires $\lceil log_2\ c - 1 \rceil$ bits to unambiguously encode the actual number. Nevertheless, we utilize $\lceil log_2\ c \rceil$ bits to encode the actual number of occupied leaf nodes, since it is only a marginal disparity but allows for a unified encoding and decoding process with techniques where no condensation is applied (that is hybrid techniques that use a quadtree as a foundation).

For the CBLQ scheme, the number of inner nodes (#in) is captured, which is $\lfloor c/3 \rfloor = i$ at a max. Therefore, $\lceil log_2\ i \rceil$ bits are required for encoding the actual number of inner nodes.

KDQT$_n^{c,a}$ The KDQT$_n^{c,a}$ method is a hybrid technique organizing the embedding space with a kd space partitioning just as KDMBR$_n^b$ but utilizing a cell-internal quadtree for the refinement of the occupied cells of the kd-structure. Therefore, a performance comparison between these two methods evaluates which refinement is more efficient for kd-structures.

Analogous to GridQT$_r^{c,a}$, the cell-interior quadtrees are built dependent on the parameters c and a and possibly with condensation. Also, the resource descriptions are structurally identical to the GridQT$_r^{c,a}$ resource descriptions.

QTMBR$_{c,a}^b$ The QTMBR$_{c,a}^b$ method is a hybrid technique organizing the embedding space with a quadtree structure—which is individual for each resource and hence a local space partitioning—as a foundation and a quantized MBR as a bounding volume for the refinement of each occupied quadtree region.

The basic quadtree is built dependent on the parameters c and a. The quantized MBR for refining quadtree regions occupied with at least one data point is built just as for KDMBR$_n^b$, the parameter b controlling the accuracy and the storage requirements. Since occupied quadtree regions are further refined, *no* quadtree condensation is applied.

The resource descriptions require incorporating some meta data beside the raw quadtree and MBR data to unambiguously decode the bit vectors:

For the LQ code, the bit vector first captures the number of levels (#lvl) or depth of the subsequent quadtree. The maximum depth or number of levels of a non-condensed quadtree with c regions is $\lfloor c/3 \rfloor = l$. Therefore, $\lceil log_2\ l \rceil$ bits are required to encode the depth of the quadtree. Afterwards, the number of occupied quadtree regions (#oqr) is captured using $\lceil log_2\ c \rceil$ bits. The raw quadtree data succeeds. Finally, the MBR data for the occupied quadtree regions is consecutively encoded (see Figure 3).

For the CBLQ scheme, the number of inner nodes (#in) of the quadtree is required as meta data and occupies the first $\lceil log_2\ i \rceil$ bins of the bit vector. Afterwards, the raw quadtree data and MBR data for the occupied quadtree regions is captured consecutively.

MBRQT^{c,a} The hybrid methods presented so far utilize approaches organizing the embedding space as a foundation. Of course, it is also practicable to utilize (a) bounding volume(s) as a foundation. Methods organizing the embedding space first need to describe the whole universe U, partitioning it into regions (not) containing data points. By using (a) bounding volume(s) as a foundation, the space partitioning can be restricted to the 'regions of interest' (that is the regions delineated by the bounding volume(s)) and does not have to take regions

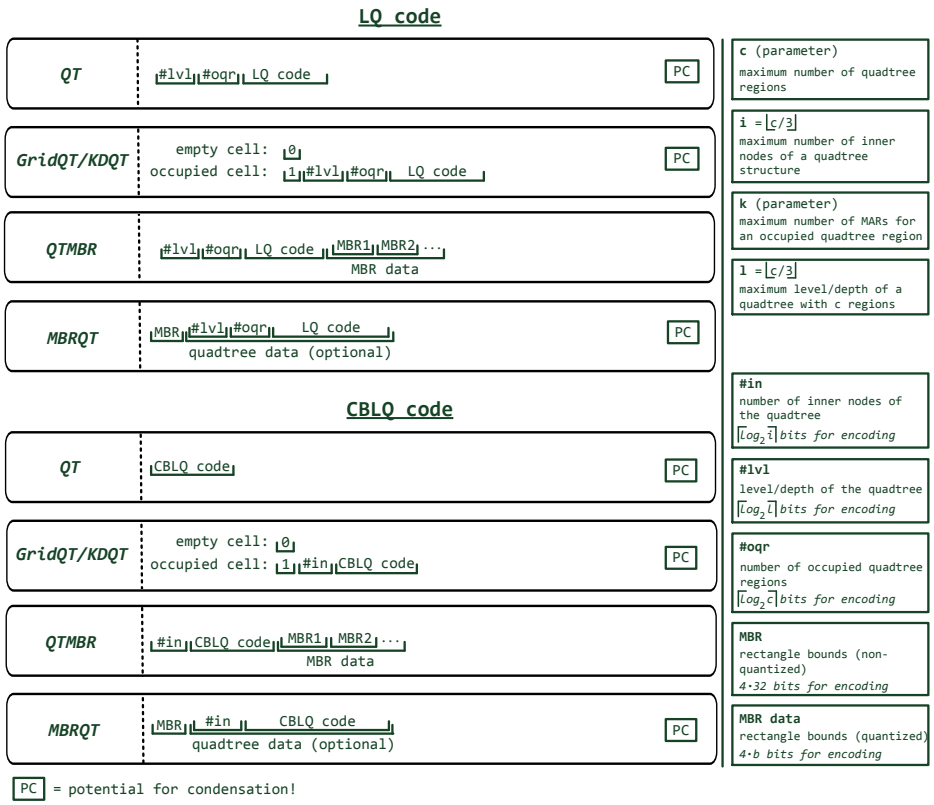


Fig. 3: Composition of the bit vectors for the different techniques and both encoding schemes.

not containing data points at all into account. The MBRQT^{c,a} method is a hybrid method combining a basic MBR as a foundation with an MBR-interior quadtree as a refinement.

First, the 'region of interest' is defined by determining the MBR of the resource's data points. The MBR-interior quadtree then is built dependent on the parameters c and a, only being bounded by the MBR and not by the universe U. Note that therefore, single quadtree regions might cross the Date Line, which is not the case for universe-related approaches such as QT$_{c,a}$ or hybrid methods using a quadtree as a foundation. The MBR-interior quadtrees can be condensed. In case the exterior MBR is a point or a very small rectangle with an area $\leq a$, no MBR-interior quadtree is built and the resource's data points are solely represented by the MBR.

The bit vector descriptions first capture the MBR bounds in single-precision floating-point format (that is $4 \cdot 32$ bits). In case the MBR area is bigger than a, they are followed by the data of the MBR-interior quadtree, which is encoded akin the cell-interior quadtrees for the GridQT$_r^{c,a}$ and KDQT$_n^{c,a}$ resource descriptions for both schemes—only missing the leading 1 for indicating an occupied cell which is unnecessary for MBRQT^{c,a}.

4 Resource Selection

With the resource descriptions presented in section 3, the problem of representing the 'content' of a resource has been elaborated. The next task is the resource selection problem.

The resource selection process has to be oriented towards the query type supported by the search system. For our search scenario and our data collection (see section 5.1), we assume precise kNN queries, i.e. the actual k nearest neighbors within the resource network shall be retrieved for a query point. This kind of resource selection process involves a ranking of the resources followed by requesting the closest data points from the resources while pruning irrelevant resources until the k nearest neighbors have been unambiguously determined.

Resource Ranking Aside from $\text{UFS}_{n,cc}$, all other resource description methods, in the very end, depict one or several rectangular areas to describe the data point cloud of a resource. Therefore, for all of them, the same ranking algorithm is applied, which utilizes the lower-bound distances from the query point to the rectangular areas. It is verbally described in the following.

> At first, the representation of each resource is constructed as a list of rectangular areas, which are built from the resource description. These representations can be cached to avoid having to rebuild them for each query. Afterwards, for each rectangular area, an `R-Entry` is built. An `R-Entry` captures the minimum distance of the rectangular area from the query point and the size of the surface area of the rectangle. The `R-Entries` of a resource are ascendingly sorted by (a) minimum distance from the query point and (b) minimum surface area (in case of equal distances). With each resource being represented by their sorted list of `R-Entries`, the actual resource ranking commences.
> A total ordering of the resources is defined by a pairwise comparison of two resources r_a and r_b, each represented by their list of `R-Entries`. The smaller list of `R-Entries` is filled with dummy entries which are most unfavorable for the ranking to avoid having unequal list sizes. To decide on the ranking between r_a and r_b, the respective `R-Entries` r_{a_i} and r_{b_i} at the same list index i are compared one after another, starting with $i = 0$. If the minimum distance of r_{a_i} is smaller than the minimum distance of r_{b_i}, r_a is ranked higher. In case of equal minimum distances, r_a is ranked higher if the surface area of r_{a_i} is smaller than the surface area of r_{b_i}. If no decision can be made by comparing the `R-Entries` at index i, i is incremented and the `R-Entries` at the next index position are compared alike. If the comparison of the entire lists of `R-Entries` does not lead to a decision, r_a is ranked higher than r_b if r_a is 'bigger' than r_b, that is administers more data points. If r_a and r_b are of the same size, a random ranking decision is made.

For $\text{UFS}_{n,cc}$, the ranking algorithm basically is the same aside from being based on the polygonial areas of the Voronoi cells instead of rectangular areas. Therefore, the Voronoi diagram has to be constructed explicitely from the sites. Only the cc closest Voronoi cells (with respect to the query point) are considered for the ranking.

The precise kNN algorithm The next step in the resource selection process is to efficiently determine the actual nearest neighbors in the resource network based on the resource ranking. In the following, we will verbally describe an algorithm for a precise kNN search, which is

implemented as an iterative range query where the query radius decreases every round. The parallelism of the search scenario is exploited by contacting n_{rp} resources per round.

At start, the query radius r is set to infinity and the `topk[]` array of the (current) k nearest neighbors is empty. First, the resources are ranked by the respective ranking algorithm for the different resource descriptions methods described above. The sorted list L_r determines the order in which the resources should be queried for their data points. The list L_r is processed until all resources from the list have been queried or successfully have been pruned, and thus the list is empty.

In each round, the resource descriptions of the next n_{rp} resources of the list L_r are examined. When the resource res cannot be pruned, the top k data points of res are requested and the global `topk[]` array is updated by replacing the (with respect to the query point q) most distant data points in `topk[]` with closer data points of the local result array of res, and finally ascendingly sorting `topk[]` by the distance from the query point again. The pruning of resources is based on lower-bound distances provided by the resource description information. If the lower-bound distance of a resource's description from the query point q is greater than the current query radius r, the resource can be safely pruned from search. For $\text{UFS}_{n,cc}$, the Voronoi cells are reconstructed in order to check which cells overlap the 'query ball' (and thus may contain relevant data points). After querying or pruning, the n_{rp} resources examined are removed from L_r and the query radius r is set to the distance of the current kth nearest neighbor for the next round. When the algorithm ends, the k nearest neighbors unambiguously have been determined. Note that after the initial ranking, no re-ranking of resources is needed.

5 Evaluation

In the evaluation, the main focus will be on two criteria: (1) The *resource description sizes* (rds). Small rds are preferable since the amount of data to be transferred in the network and the general storage space requirements are reduced. We will concentrate on the resource description sizes averaged over the set of resources. (2) The *resource fraction contacted* (rfc) in order to retrieve the top k data points in the resource network. The fewer resources need to be queried for their data points, the better the resource descriptions.

Obviously, the optimization of the two criteria is conflicting: the more storage space is spent, the more accurate the resource descriptions can be and fewer resources typically need to be queried; analogous the reverse. Varying the parameterization of the different techniques influences the accuracy of the resource descriptions and hence the rds and the rfc values. We apply the Skyline operator [BKS01] to filter out the 'interesting' parameterizations for a specific technique. Considering the measurements for the different techniques and their parameterizations as sets of two-dimensional points, with rds being the x-dimension and rfc being the y-dimension, a point is 'interesting' if it is not dominated by any other point. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension [BKS01, p.1]. Thus, for a specific technique, a point p_x representing the parameterization x dominates a point p_y representing a parameterization y if ($p_x.rfc \leq p_y.rfc$ and $p_x.rds < p_y.rds$) or ($p_x.rfc < p_y.rfc$ and $p_x.rds \leq p_y.rds$).

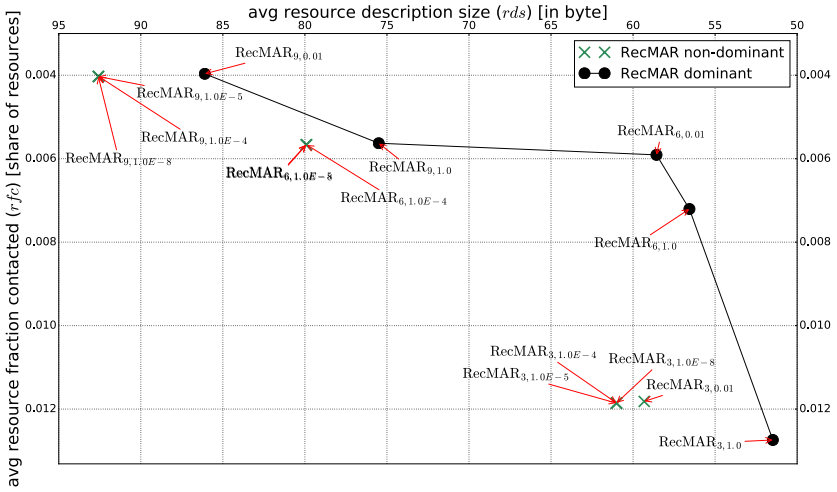


Fig. 4: Exemplary Skyline determination for RecMAR$_{k,sl}$. The black dots connected by the black lines forge the Skyline of the dominant parameterizations for RecMAR$_{k,sl}$.

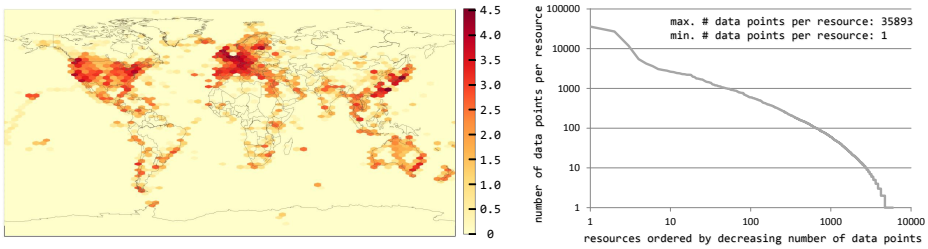


Fig. 5: Distribution of the data points in the data space (left) and number of data points per resource (right). Note that the values on the left legend are $log_{10}(x+1)$ scaled, so the number of data points per bin is calcluated by $x = 10^n - 1$. For example, $n = 4.0$ results in $x = 9,999$.

Figure 4 illustrates the Skyline determination for the RecMAR$_{k,sl}$ technique. Note that both axes are inverted, i.e. the more northeast-bound a point, the more dominant it is. The Skylines determined for the respective techniques will be used for inter-technique comparisons.

5.1 Experimental Setup

The data collection is based on 406,450 geo-referenced images uploaded to Flickr by 5,951 different users. The images were crawled in 2008. The corresponding spatial data points are assigned to resources via user ID. Hence, it is assumed that every user in the network operates an own resource for spatial data. The spatial distribution of the data points and the distribution of the amount of data points maintained by the resources are depicted in Figure 5. The spatial distribution is very patchy and the distribution of data points to resources is skewed: few resources maintain a lot of data points and a lot of resources maintain only a few data points. The skewed distribution is typical for Peer-to-Peer (P2P) data [Cu03, p. 5].

MBR		no parameters
$RecMAR_{k,sl}$	k	3|6|9
	sl	1.0|0.01|1.0E-4|1.0E-5|1.0E-8
$UFS_{n,cc}$	n	512|2048|8192
	cc	16|64|256|n
$KDMBR^{b}_{n}$	n	512|2048|8192
	b	3|4|6|8
$QT_{c,a}$	c	256|512|1024|2048|8192
	a	0.001|1.0E-5|1.0E-7|1.0E-8
$GridQT^{c,a}_{r}$	r	16|32|64
	c	16|32|64|256|512
	a	0.001|1.0E-5|1.0E-7|1.0E-8
$KDQT^{c,a}_{n}$	n	512|2048|8192
	c	16|32|64|256|512
	a	0.001|1.0E-5|1.0E-7|1.0E-8
$QTMBR^{b}_{c,a}$	c	64|256|512|1024
	a	0.1|0.05|0.01|0.001
	b	3|4|6|8
$MBRQT^{c,a}$	c	16|64|256|512|1024|2048|8192
	a	0.001|1.0E-5|1.0E-7|1.0E-8

a = stopping **a**rea

b = # of **b**its

c = # of **c**ells

cc = # of **c**entroids **c**onsidered

k = # of MARs

n = **n**umber of subspaces

r = # of **r**ows

sl = thre**s**ho**l**d distance rectangle center vs. associated data points

Note that for the hybrid techniques, the parameters of the foundation are always subscript whereas the parameters for the refinement are always superscript.

Fig. 6: Listing of the tested parameter variations for the different techniques.

For the evaluation, we assume a kNN search for the k nearest spatial data points—associated with media content such as images—with respect to a spatial query location (or query point) in a resource network. It is assumed that every resource knows the resource description of every other resource in the network. The query points are determined as follows: At first, a random resource is selected; then, a random data point from the resource is chosen as a query point. This simulates that all resources—irrespective of the size—have the same probability to issue a query. For the 50 selected query points, 7.74 resources contribute to the top 50 data points on average. This results in an rfc of $7.74/5,951 = 0.0013$ for an ideal resource selection technique which only contacts relevant resources for their data.

The parameters of the techniques are varied in certain windows which seem suitable in terms of run time for the summary computation and general storage space requirements (see Figure 6). Also, the GPS accuracy is taken into account for the quadtree-based techniques so that the stopping area parameter matches the GPS accuracy limits on the highest parameterization. All possible parameter combinations are tested.

Some resource description techniques rely on randomness; for example $UFS_{n,cc}$ (selection of the sites) or the quadtree-based techniques (selection of the next quadtree region to split when there are several equally sized black regions). For these techniques, we conducted four runs with different seeds to minimize the effect of outliers for each parameterization (using the same set of 50 query points in each run).

5.2 Optimizations

To augment the capability of the search system, we conduct several optimizations which affect the space efficiency of the resource descriptions.

The resource descriptions must be serialized to be distributed in the network. This requires 27 byte serialization overhead, which are included in the measurements (see section 5.3). We applied two optimizations to reduce the storage requirements of the resource descriptions:

Since all resource descriptions in the very end are encoded as bit vectors, we attempt to compress them by using `Javas`'s `gzip` implementation with default parameters. In case the compression turns out to result in a reduced amount of data, the compressed bit vector will be distributed in the network.

For 'small' resources—and depending on the technique and its parameterization—the resource descriptions might need more space than transferring the few data points directly. A direct transfer of course is also more precise, since the summarizations of the resource data only describe approximated areas containing data points. Therefore, in case the size of the resource descriptions equals or exceeds the storage space needed to encode the data points themselves, the resource 'directly' is represented by the data points it maintains rather than by an approximated description of the areas where the data points are located.

All in all, for non-quadtree-based methods, there are four ways of transferring the data representing the resource (the *resource description*), depending on if the resource is described by approximating areas (the resource *summary*) or directly by its data points (the *direct representation*), and whether the data is compressed or not. For quadtree-based *summaries*, it additionally has to be differentiated if the quadtrees are encoded by the LQ scheme or the CBLQ scheme, resulting in six different resource description types. The information on the resource description type has to be incorporated into the resource description. For a unified encoding scheme, we utilize 3 bits for all resource descriptions. Additionally, the resource sizes which are used in the ranking algorithms are encoded with 5 bit, overall resulting in 1 extra byte in addition to the serialization and the resource description data itself.

Note that the resource ranking is not affected by the compression but by the direct transfer: resources, for which the data points directly have been transferred, are not ranked since the exact locations of their data points already are known. The possibility of transferring the data points itself rather than a summary also has consequences for the query processing and the determination of the rfc values. The query is processed in a two-step approach:

In the first step, the coordinates of the top k closest data points w.r.t. the query point are determined. The coordinates of the data points which have been directly transferred are already known (therefore, the corresponding resources do not have to be contacted in this step); for the coordinates of the remaining data points (approximated by the summaries), the respective resources need to be contacted.

In the second step, all resources contributing to the top k data points are contacted for the media data associated with these data points.

Hence, the rfc value arises as result from: $(res_{sum} + res_{dt})/res$, with res_{sum} being the number of ranked resources (i.e. resources which transferred their summaries as resource descriptions) contacted while the kNN algorithm is applied, res_{dt} being the number of resources which directly transferred their data points and contribute to the top k result and res being the cardinality of the resource set.

5.3 Experimental Results

The listed baseline in Figure 7 is the minimum number of resources that need to be contacted. The results show that it is possible to eminently surpass the selectivity of an MBR while expending only little extra storage. With only 1.7 byte additional storage on average (rds value), $\text{QTMBR}^3_{64,0.1}$ achieves a more than five times better selectivity (lower rfc value) (MBR: rds = 42.34 byte, rfc = 0.0471; $\text{QTMBR}^3_{64,0.1}$: rds = 44.04 byte, rfc = 0.0087).

The uniform grid obviously is not a suitable foundation, since it is the worst hybrid technique by quite a margin. Even $\text{MBRQT}^{c,a}$, despite being based on the subpar MBR as foundation, completely dominates $\text{GridQT}^{c,a}_r$ by its Skyline. For the Space Partitioning Approaches, $\text{UFS}_{n,cc}$ outperforms the solitary $\text{QT}_{c,a}$. Comparing the Skylines, the gap seems to grow with increasing rds. Considering the additional complexity (the need of a selection mechanism for the sites and their deployment within the network) plus the considerably lower entropy of the $\text{UFS}_{n,cc}$ summaries due to longer zero sequences in the bit vectors (meaning the `gzip` entropy compression works better since quadtree data generally is of high entropy), the results for $\text{QT}_{c,a}$ are still decent in comparison, though.

The assessment of the results for $\text{QTMBR}^b_{c,a}$ (all necessary information already contained in the resource descriptions) versus state-of-the-art KDMBR^b_n (the fundamental space partition needs to be spread separately) is similar. For rds values of about 55 byte, $\text{QTMBR}^b_{c,a}$ even is superior to KDMBR^b_n. Assuming the typical 'knee-like' pathway of the Skylines, this would also hold for smaller rds values, but cannot be confirmed due to the delayed advent of the KDMBR^b_n Skyline. The KDMBR^b_n Skyline completely dominates the $\text{KDQT}^{c,a}_n$ Skyline. Therefore, in case the foundation is already decently accurate, quantized MBRs are a more efficient means of refinement compared to internal quadtrees. For coarse foundations like the uniform grid, we expect internal quadtrees to be more suitable. $\text{QTMBR}^b_{c,a}$ almost completely dominates its 'mirror-technique' $\text{MBRQT}^{c,a}$—especially for low rds—on a large margin. Combining (adaptive) space partitioning with quantized bounding volumes seems to be more promising than utilizing full accuracy bounding volumes, be it hybrid or solitary. This also backed by $\text{RecMAR}_{k,sl}$ being clearly distanced by the other techniques from about 62 bytes rds on, only dominating $\text{GridQT}^{c,a}_r$ before. Generally, at the latest from about 90 bytes rds on, only marginal differences between the different techniques can be observed (all of them conducting an asymptotic approximation towards the rfc baseline).

For the data transfer option chosen, the share of directly transferred data points as well as the share of zipped summaries increases with increasing rds values, which was to be expected. In Table 1, for each technique, we display the margin in which different key figures vary for the Skyline-forming parameterizations. Hybrid techniques using global space partitioning as a foundation and also $\text{UFS}_{n,cc}$ benefit a lot from compression, their share of zipped summaries generally is high. The foundation seems to be decisive regarding the entropy, since the share of zipped summaries is high even when internal quadtrees are utilized as a refinement. In contrast, hybrid techniques using quadtrees as a foundation as well as the solitary $\text{QT}_{c,a}$ yield a significantly lower share of zipped summaries. Thus, both overhead for distributing the information about the global space partitioning as well as for (de-)compression can be avoided when using these techniques. In general, the share of

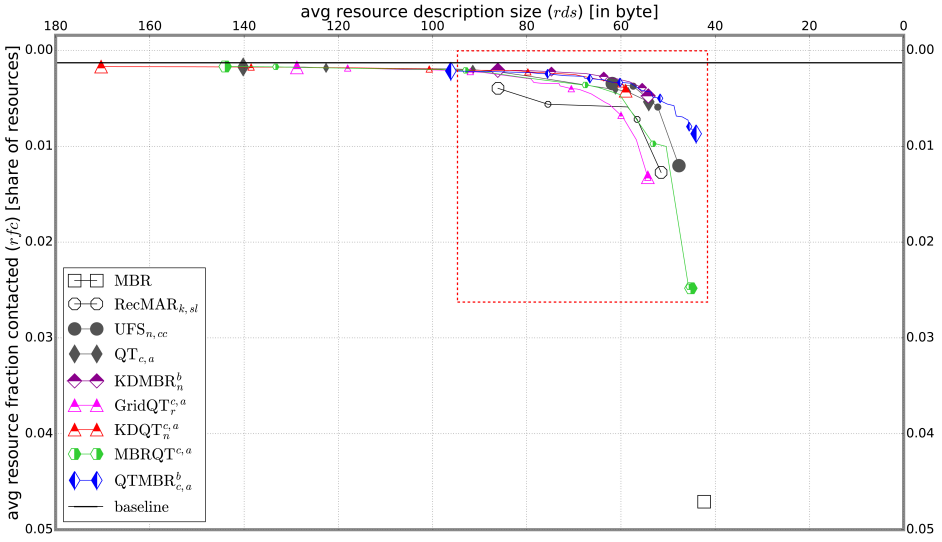


Fig. 7: Overview of the Skyline comparison for the different techniques. Markers on the Skylines are sampled, i.e. not all rds/rfc data points which constitute a Skyline are depicted by a marker.

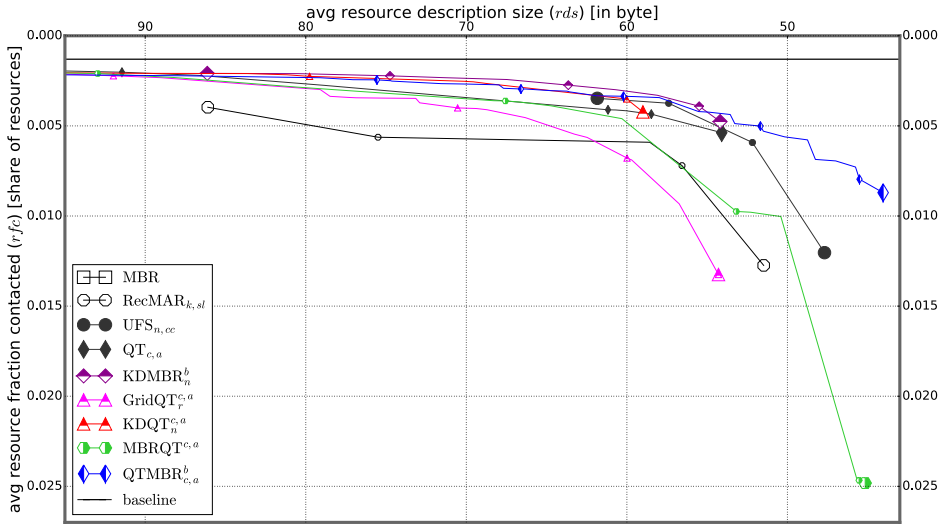


Fig. 8: Detailled view on the experimental results (magnification of the red rectangle in Figure 7).

CBLQ-coded quadtrees is higher the more the parameterization fosters the formation of large quadtrees. Thus, the LQ scheme is more effective for encoding small quadtrees but becomes inefficient for larger quadtrees, which is intuitive since the LQ scheme encodes the path for every black node separately (which imposes redundancy for larger quadtrees).

Tab. 1: Spans in which several key figures vary for the parameterizations forming the Skyline of the respective techniques. Remember that for techniques not utilizing quadtrees, there is no choice between the LQ scheme and the CBLQ scheme for the summary. Thus, the only option is to (not) zip them; the respective shares are displayed in the `lqz/sumz` or `lqnz/sumnz` rows. `lq(n)z`: (non-)zipped LQ scheme summaries, `sum(n)z`: (non-)zipped 'non-quadtree' summaries, `dt(n)z`: (non-)zipped directly transferred data points, `cblq(n)z`: (non-)zipped CBLQ scheme summaries.

technique → / key figures ↓	MBR	RecMAR$_{k,sl}$	UFS$_{n,cc}$	KDMBR$_n^b$	QT$_{c,a}$	GridQT$_r^{c,a}$	KDQT$_n^{c,a}$	QTMBR$_{c,a}^b$	MBRQT^{c,a}
lqz/ sumz	0%	0.05% - 15.2%	47.0% - 63.5%	48.5% - 59.4%	0% - 0.48%	12.5% - 55.5%	18.9% - 52.5%	0%	0% - 1.1%
lqnz/ sumnz	70.7%	40.1% - 67.1%	0.26% - 23.2%	1.0% - 18.6%	26.2% - 69.6%	0% - 2.3%	0.48% - 16.1%	42.3% - 82.7%	38.2% - 51.4%
dtz	0%	0.05% - 0.72%	0% - 1.4%	0.02% - 2.1%	0% - 0.15%	0.03% - 8.1%	0.03% - 7.0%	0% - 0.03%	0% - 0.18%
dtnz	29.2%	32.8% - 44.0%	29.8% - 37.8%	30.6% - 41.3%	0% - 40.9%	35.5% - 52.7%	32.3% - 51.5%	0% - 29.6%	29.2% - 41.1%
cblqz	-	-	-	-	0% - 25.1%	8.2% - 27.3%	5.8% - 18.4%	0% - 1.0%	0% - 15.8%
cblqnz	-	-	-	-	7.6% - 30.4%	0% - 0.75%	0.07% - 3.8%	17.3% - 28.8%	3.8% - 22.6%
avg. res. desc. size (in byte)	42.3	51.5 - 86.1	47.7 - 61.8	54.2 - 86.1	54.1 - 140.2	54.3 - 128.8	59.0 - 170.3	44.0 - 96.1	45.2 - 144.0
max res. desc. size (in byte)	44	76 - 172	90.25 - 307	202 - 2299.5	113 - 1983.5	216.75 - 5439.5	369 - 9934.5	82.75 - 2079	50 - 2071.25

The RecMAR$_{k,sl}$ Skyline is built by parameterizations of $sl = 1$ or 0.01. Apparently, dividing the point sets into very narrow groups and allocating a full precision rectangle to bound each of them is not worthwhile. The dominant UFS$_{n,cc}$ parameterizations all consider a small amount of cells in the ranking process ($cc = 16$). The cell occupancy of distant cells thus is negligible concerning the relevance of a resource for the 'region of interest', the size of the resources proves more relevant (low cc values benefit bigger resources, see section 4). For KDMBR$_n^b$, almost all parameterizations are located on the Skyline, solely the use of $b = 3$ for the quantized rectangles is mostly not sufficient. The Skyline of GridQT$_r^{c,a}$ is composed of few parameterizations with $r = 64$. Hence, also for an intra-GridQT$_r^{c,a}$ consideration, the adaptivity of the space partitioning should be added in time. For KDQT$_n^{c,a}$, the share of non-zipped summaries is 15% at least when $n = 512$. For more cells, the shares rapidly drop. The maximum rds is much bigger compared to GridQT$_r^{c,a}$. This is because the adaptivity will result in very many cells being occupied in the extreme case—with each cell featuring an internal quadtree for refinement. The Skyline of MBRQT^{c,a} misses a lot of parameterizations with 'middle-sized' internal quadtrees ($c = 256$ to 1024); hence, the use of a small or a large amount of cells seems to be way to go.

Small quadtrees can be encoded very efficiently: for $QT_{256,0.001}$, not a single resource directly transfers its data points (69.6% LQ encoded, 30.4% CBLQ encoded). Generally, for $QT_{c,a}$, small values for the stopping area a only become efficient with a high value for c; for quadtrees with a low amount of cells (small c value), a bigger stopping area is more suitable. Only 4 out of 34 parameterizations on the $QTMBR^b_{c,a}$ Skyline utilize $b = 8$; the use of $b = 4$ or $b = 6$ seems to be most suitable for $QTMBR^b_{c,a}$. $QTMBR^b_{c,a}$ displays the lowest share of zipped summaries and therefore benefits least from compression. This is most remarkable, since $QTMBR^b_{c,a}$, even for bigger rds, is very competitive to techniques strongly benefitting from compression.

6 Related Work

The techniques presented in this paper are based on or strongly related to multidimensional access methods supporting search operations in centralized databases. Within these, it is distinguished between Point Access Methods (PAMs), for searching sets of points in two or more dimensions, and Spatial Access Methods (SAMs), which handle spatially extended objects. See [Sa05] and [GG98] for an extensive overview on these topics. Both PAMs and SAMs are applied in rather low-dimensional data spaces which are coordinate-based. For high-dimensional data spaces not based on coordinate systems, Metric Access Methods (MAMs) have been developed. See [He09] for an extensive tutorial on MAMs. Hierarchical data structures akin the quadtree are used in numerous application fields. See [Sa84] for a fundamental survey. Manouvrier et al. illustrate several possibilities for the linear storage of quadtrees in [MRJ02].

7 Conclusion

Hybrid Approaches for spatial resource description show higher potential for describing the geospatial footprint of resources compared to solitary techniques (both Geometric Approaches and Space Partitioning Approaches)—a suitable selection of techniques to be combined presumed. Within these, quadtree-based techniques are very competitive and—in particular a combination of a quadtree and quantized cell-interior MBRs ($QTMBR^b_{c,a}$)—provide similar performance to the state-of-the-art ($KDMBR^b_n$). Furthermore, they simultaneously encode all necessary information within the resource descriptions, therefore superseding the need to separately sample information about the data collection as a whole plus processing the collected data and distributing the result in the network afterwards, considerably reducing the complexity of the search system.

References

[Be92] Becker, B.; Franciosa, P. G.; Gschwind, S.; Ohler, T.; Thiemt, G.; Widmayer, P.: Enclosing many boxes by an optimal pair of boxes. In: *Proc. of STACS 92: 9th Ann. Symp. on Theor. Aspects of Comp. Sc. Cachan, France*. Springer Berlin Heidelberg, pp. 475–486, 1992.

[BH12] Blank, D.; Henrich, A.: Describing and Selecting Collections of Georeferenced Media Items in Peer-to-Peer Information Retrieval Systems. In: *Discovery of Geospatial Resources: Methodologies, Technologies, and Emergent Applications*. Information Science Reference, pp. 1–20, 2012.

[BHK16] Blank, D.; Henrich, A.; Kufer, S.: Using Summaries to Search and Visualize Distributed Resources Addressing Spatial and Multimedia Features. Datenbank-Spektrum 16/1, pp. 67–76, 2016.

[BKS01] Börzsönyi, S.; Kossmann, D.; Stocker, K.: The Skyline Operator. In: *Proc. of the 17th Int. Conf. on Data Engineering*. IEEE Computer Society, Washington, DC, USA, pp. 421–430, 2001.

[Ca00] Callan, J.: Distributed Information Retrieval. In: *Advances in Information Retrieval*. Kluwer Academic Publishers, pp. 127–150, 2000.

[Ca05] Caldwell, D. R.: Unlocking the Mysteries of the Bounding Box. A/2, pp. 1–20, Aug. 2005.

[Cu03] Cuenca-Acuna, F. M.; Peery, C.; Martin, R. P.; Nguyen, T. D.: PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In: *12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12 '03)*. IEEE Press, Seattle, Washington, pp. 1–11, 2003.

[Ga82] Gargantini, I.: An Effective Way to Represent Quadtrees. *Commun. ACM* 25/12, pp. 905–910, Dec. 1982.

[GG98] Gaede, V.; Günther, O.: Multidimensional Access Methods. *ACM Comput. Surv.* 30/2, pp. 170–231, 1998.

[HB10] Henrich, A.; Blank, D.: Description and Selection of Media Archives for Geographic Nearest Neighbor Queries in P2P Networks, 2010.

[He09] Hetland, M. L.: The Basic Principles of Metric Indexing. In: *Swarm Intelligence for Multi-objective Problems in Data Mining*. Springer Berlin Heidelberg, pp. 199–232, 2009.

[KBH12] Kufer, S.; Blank, D.; Henrich, A.: Techniken der Ressourcenbeschreibung und -auswahl für das geographische Information Retrieval. In: *Proc. of the IR Workshop at LWA 2012*. Dortmund, Germany, pp. 1–8, 2012.

[KBH13] Kufer, S.; Blank, D.; Henrich, A.: Using Hybrid Techniques for Resource Description and Selection in the Context of Distributed Geographic Information Retrieval. In: *Advances in Spatial and Temporal Databases: 13th Intl. Symp., SSTD 2013, Munich, Germany*. Springer Berlin Heidelberg, pp. 330–347, 2013.

[KH14] Kufer, S.; Henrich, A.: Hybrid Quantized Resource Descriptions for Geospatial Source Selection. In: *Proc. of the 4th Int. Workshop on Location and the Web*. LocWeb '14, ACM, Shanghai, China, pp. 17–24, 2014.

[Li97] Lin, T.-W.: Set Operations on Constant Bit-length Linear Quadtrees. *Pattern Recogn.* 30/7, pp. 1239–1249, July 1997.

[MRJ02] Manouvrier, M.; Rukoz, M.; Jomier, G.: Quadtree representations for storage and manipulation of clusters of images. *Im. Vis. Comp.* 20/7, pp. 513–527, 2002.

[Oo99] Oosterom, P. V.: Spatial Access Methods. In. Vol. 1, *Geographical Information Systems*, chap. 27, pp. 385–400, 1999.

[Sa05] Samet, H.: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Sa84] Samet, H.: The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16/2, pp. 187–260, June 1984.

[SK90] Seeger, B.; Kriegel, H.-P.: The Buddy Tree: An Efficient and Robust Access Method for Spatial Data Base. In: *Proc. of the Sixteenth Intl. Conf. on VLDB*. Morgan Kaufmann Publishers Inc., Brisbane, Australia, pp. 590–601, 1990.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 465

Efficient Z-Ordered Traversal of Hypercube Indexes

Tilmann Zäschke[1], Moira C. Norrie[2]

Abstract: Space filling curves provide several advantages for indexing spatial data. We look at the Z-curve ordering and discuss three algorithms for navigating and querying k-dimensional Z-curves. In k-dimensional space, a single hyper-'Z'-shape in a recursive Z-curve forms a hypercube with 2^k quadrants. The first algorithm concerns efficient checking whether a given quadrant of a local hyper-'Z' intersects with a global query hyper-box. The other two algorithms allow efficient Z-ordered traversal through the intersection space, either based on a predecessor from inside the intersection (second algorithm) or from any predecessor (third algorithm). The algorithms require an initialisation phase of $\Theta(k)$ for encoding the intersection envelope of the local hyper-'Z' with a range query. Using this envelope, all three algorithms then execute in $\Theta(1)$. The algorithms are however limited by the register width of the CPU at hand, for example to $k < 64$ on a 64 bit CPU.

Keywords: Multi-dimensional index, spatial index, PH-tree, binary hypercube, Z-curve, Z-ordering, window queries

1 Introduction

We discuss the problem of efficiently traversing partitions of binary hypercubes. This problem occurs for example in certain multi-dimensional indexing structures when executing window queries. Part of the problem is to efficiently determine for increasing dimensionality k whether points or regions in the trees intersect with a k-dimensional query hyper-box. In the case of indexing structures such as the UB-tree [Ba97, Ma99], the BUB-tree [Fe02], the theoretical HQ-tree [Sk06], or the more recent BQ-tree [ZYG11] or PH-tree [ZZN14], the multidimensional data is often (partially) interleaved to form bitstrings. Depending on the encoding, these bitstrings impose a natural Z-order on the stored keys, hence they are called *Z-addresses*. When a Z-address is sliced into sub-strings of k bits, each slice can be seen as the address (called H-address) of a quadrants of a k-dimensional binary hypercube. If we define that every quadrant can contain at most one child (sub-node or point) then nodes may often have up to 2^k children[3] and every child has its own H-address.

Window queries are usually implemented as a type of node traversal while avoiding nodes and their children if they cannot potentially intersect with the query box. To perform efficient window queries, it is therefore necessary to traverse all necessary nodes and sub-nodes while efficiently avoiding any unrelated nodes. Just as the tree is represented by a hierarchy of nodes, i.e. binary hypercubes, each query box can be seen as the envelope of all potentially matching values and thus can also be seen as a (convex) hierarchy of binary

[1] ETH Zurich, Department of Computer Science, Universitätsstrasse 6, 8092 Zurich, zaeschke@inf.ethz.ch
[2] ETH Zurich, Department of Computer Science, Universitätsstrasse 6, 8092 Zurich, norrie@inf.ethz.ch
[3] The advantage of such large nodes may not be intuitive but has been explained and demonstrated in [ZZN14, Zä15, Ch15, BCA15, LH14].

hypercubes. Window queries therefore represent the problem of intersecting hierarchies of binary hypercubes and then traversing these intersections. Related to the window queries are some algorithms for skyline [BKS01] queries, i.e. queries for non-dominated vectors. Some of the more recent algorithms [Ch15, BCA15, LH14] partition the space hierarchically around pivot points and then use H-addresses to reduce the search space.

In this paper, we discuss three algorithms for efficiently navigating intersections of k-dimensional binary hypercubes. All three algorithms can be used for traversing intersections, but they exhibit different strength depending on the representation of the hypercube in memory and on the size of the intersection compared to the size of the hypercube. The 1st algorithm has been used before but without explanation [ZZN14] or only partial explanation in [Ch15, BCA15]. The 2nd and 3rd algorithm are contributions of this paper.

The first algorithm **isInI**(h, I) checks whether a H-address h lies in the intersection I of a node N with a query hyper-box. This is useful when we traverse the whole hypercube $\mathbb{H}^k$, for example if $|I|$ is approaching $|\mathbb{H}^k| = 2^k$.

The second algorithm **inc**$(h_{I,a})$ takes a H-address $h_{I,a} \in I$ and returns its successor $h_{I,b} \succ h_{I,a}$ in I. This allows efficient traversal of I if $|I| \ll 2^k$ by omitting any $h \notin I$.

The third algorithm **succ**(h) takes any H-address h and returns the next smallest $h_I \succ h$ that lies in I. **succ**(h) is a generalisation of **inc()** that accepts any $h \in \mathbb{H}^k$ which can be useful for large hypercubes where it can make sense to switch between the first and second algorithm.

All algorithms require encoded intersection information which can be calculated in $\Theta(k)$ for each binary hypercube of the tree. Using this intersection information, all three algorithms complete in $\Theta(1)$ on a CPU with at least $(k + 1)$ bits per register, for example for $k < 64$ on a 64 bit CPU. This differentiates us from earlier work which usually processes only one bit at a time, for example [KPS91].

Algorithms similar to **isInI**(h, I) have already been presented in research [ZZN14, Ch15, BCA15, LH14] where they are used to skip quadrants of the binary hypercube. However, while using **isInI**(h, I) is efficient if only few quadrants have to be skipped, the algorithm does not scale well with larger dimensionality k where only a minority of exponentially growing number of quadrants should be visited. For these cases we present the novel algorithms **inc**$(h_{I,a})$ and **succ**(h) which allow jumping directly to the next valid quadrant.

This paper is structured as follows. After presenting related work in Sect. 2 we define terminology in Sect. 3. Sect. 4 describes how I is calculated, followed by the sections 5, 6 and 7 which describe the three algorithms. After that, in Sect. 8, we provide a discussion how and when the algorithms could be used, followed by a concluding discussion in Sect. 9.

2 Related Work

Window queries on Z-curves [OM84] are relevant for efficient implementations of multi-dimensional indexes such as the Universal B-tree (UB-tree) [Ba97, Ma99, Ra00], the

BUB-tree [Fe02] or the PH-tree [ZZN14, Zä15][4], see for instance [Sa06] for other examples. As proposed by [TH81], these trees interleave some or all bits of each dimension of a stored k-dimensional point $p = p_0, p_1, ...p_{k-1}$ into one bit string. We call this bitstring *Z-address* z since it can be seen as a coordinate in a space-filling Z-curve, i.e. the lexicographic ordering of values encoded in Z-addresses is the Z-ordering, see [Sa94] for a discussion. A comparison of Z-curves with other space filling curves is given in [MAK03].

In the following we assume for simplicity that all attributes p_i of an entry p have the same number of bits w, however the algorithms can be adapted to values with different lengths with few changes. For a k-dimensional tree with Z-ordering we can split the Z-address into w chunks of k bits called *H-addresses* h which represent addresses in binary hypercubes. In effect, an H-address h_v is a cross-section through all $p_{0 \leq i < k}$ of a point p, taking the vth bit from each p_i.For example, a 2-dimensional point $p = p_0, p_1$ with $p_0 = 100$ and $p_1 = 010$ can be interleaved to a Z-address $z = 100100$. z is then sliced into $w = 3$ chunks of $k = 2$ bits each where the three chunks represent the H-addresses in the binary hypercube of the root node $h_{d=0} = 10$, an inner node $h_{d=1} = 01$ and a leaf node $h_{d=2} = 00$.

In each binary hypercube, we define H-addresses $h \in \mathbb{H}^k = \{0, 1, ..., 2^k - 1\}$ as the lexicographically ordered sequence of points which, when each connected with their predecessor and successor, form a k-dimensional binary Z-curve. For example, lets assume a tree consisting of a hierarchy of nodes where each node represents a hypercube. In every of the tree's hypercubes/nodes, if a quadrant of a hypercube contains geometrically more than one of the points in the tree, then the quadrant simply references a sub-hypercube/node of the same size as the quadrant. These sub-hypercubes split the space recursively further until every quadrant contains at most one point.

For window queries, the query hyper-box (the term *box* denotes that it is not necessarily a cube) is intersected with the hypercubes of the nodes, where the quadrants of the nodes are identified by H-addresses. The idea is that when we search for entries that match the query, we want to traverse only those H-addresses/quadrants in a node that intersect with the query hyperbox and who can potentially contain matching sub-nodes or points.

The naive approach is to iterate through all non-empty quadrants in a node and test whether they intersect with the query. How the H-address of a quadrant can be checked for intersection in constant time is described in [ZZN14]. This works well if the intersection I occupies a large part of the node's hypercube. For $k = 2$ it cannot be smaller than $\frac{|I|}{\mathbb{H}^k} \geq \frac{1}{2^k} = \frac{1}{4}$, but for growing k and small selectivity the naive approach becomes exponentially inefficient. An improved version with min/max quadrants (see Sect. 4) is described in [Ni13]. However, they still appear to check all quadrants between min and max in order to skip them.

In this paper we propose an approach that, instead of iterating through all available quadrants, generates only H-addresses that intersect with the query and then checks whether the H-address exists in the hypercube of the current node. The algorithm requires $\Theta(k)$ initialisation effort for each visited node and then generates only H-addresses that are part of the intersection in $\Theta(1)$ for each H-address on a CPU with more than k bits per register.

[4] [ZZN14] uses the term 'range queries' for queries on rectangular windows.

More recently, binary hypercube partitioning is also used by skyline algorithms [BKS01], i.e. algorithms that determine all non-dominated vectors in a dataset. Unlike the index trees above, they usually do not split the space in half according to the binary representation of the coordinates, but instead split at specially calculated 'pivot' points [Ch15, BCA15, LH14]. However they still use H-addresses and partially describe algorithms similar to our initial **isInI**(h, I) algorithm. Their description and use of **isInI**(h, I) is partial in the sense that they only consider cutting away the 'upper' half of the cube in each dimension. As discussed in Sect. 6, cutting away the lower half works mostly, but not completely, symmetric.

3 Terminology

In this work we refer to *constant time* operation as anything that can be executed on a computer's CPU in a constant amount of CPU operations. For example, all $\Theta()$ and $O()$ references refer to execution complexity on a CPU with at least $(k + 1)$ bits per register, unless stated otherwise.

Definition 1. *(Point p) A point p represents an entry in a tree structure. p is a k-dimensional point where each dimension is represented by a value $p_{0 \le i < k}$ with w bits, i.e. each value p_i is an integer with $0 \le p_i < 2^w$. For simplicity we ignore negative integers and floating point values, even though they are also supported by the algorithms, possibly with minor modifications[5]. We also assume that all values p_i have the same number of bits w.*

Definition 2. *(Z-address z) A Z-address z is a bitstring consisting of the $k \times w$ bits of a point p. The first k bit of z represent the first (highest order) bit of each value p_i, the next k represent the second bit, and so forth.*

Definition 3. *(H-address h) Let $h \in \mathbb{H}^k = \{0, 1, ..., 2^k - 1\}$. A hypercube address h is any subsequence of k bits starting at a multiple of k of a bitstring z. Any z thus can be seen as a sequence of w H-addresses which designate a point p in the hierarchical hypercube of the index tree. Inside the node of a tree, each h acts as a key to a point that is stored in the node or to a subnode.*

Definition 4. *(Sets I, N and R) $N \subseteq \mathbb{H}^k$ denotes the set of all h that are stored in a node, either in the form of points or subnodes. $I \subseteq \mathbb{H}^k$ is the set of h that potentially contains points or subnodes that intersect with the query box. $h \in I$ do not necessarily exist, i.e. I may or may not be a subset of N. Finally, $R \subseteq \mathbb{H}^k$ is the result set containing all h that lie in I and in N, i.e. $R = I \cap N$.*

We denote the bitwise binary operations as follows: '&' (AND), '|' (OR), '~' (NOT) and '⊕' (XOR). The listings use the same notation except ^ for ⊕.

4 Encoding the Shape of *I*

In order to efficiently traverse an intersection I, we use an efficient encoding of the shape of I. We do this by means of two bit sequences, m_0 and m_1. m_0 and m_1 each consist of k bits

[5] See [ZZN14] for a related discussion.

where each bit in m_0 specifies whether the '0'-half of that dimension of the hypercube is part of I or not. Accordingly, m_1 specifies the '1'-half of the dimension.

Definition 5. *(Range filters m_0, m_1). We encode the intersection I of the hypercube, with the query box in two bit strings, m_0 and m_1. For each dimension $i : 0 \leq i < k$, m_0 has a '0' at position i iff the '0' quadrant of that dimension is part of the intersecting body. Accordingly, m_1 has a '1' at position i iff the '1' quadrant of dimension i is part of the intersection.*

Definition 6. *($m_{0,i}$ & ~$m_{1,i} \equiv 0$). We define that m_0 and m_1 never restrict on the same dimension. In other word, m_0 and ~m_1 never have a '1' bit at the same position. If they would restrict on the same dimension, neither '0' nor '1' would be allowed for $h \in I$ at that position and the intersection $I = \emptyset$ would be empty. In this case, the current node should have never been entered because it can not possibly contain any results.*

Corollary 1. *($m_0 \leq m_1$). $m_0 \leq m_1$ follows implicitly from Def. 5 and Def. 6 because each bit in m_0 is necessarily smaller or equal to the according bit in m_1. If m_0 is bitwise smaller or equal to m_1, then $m_0 \leq m_1$.*

Corollary 2. *(m_0, $m_1 \in I$ and $\forall h : (h \in I \rightarrow m_0 \leq h \leq m_1)$). m_0 and m_1 are valid H-addresses with $m_0, m_1 \in I$. If we were to construct a minimum value $h_{min} \in I$, we would set all bits to '0', except those bits that are required to be '1'. This is identical to how we construct m_0, therefore $m_0 = h_{min}$ and hence $m_0 \in I$. The argument for m_1 goes analogous. While m_0 and m_1 define the minimum and maximum values for h through lexicographic ordering, not all values $h \in [m_0, m_1]$ are in I, i.e. $h \in I \rightarrow h \in [m_0, m_1]$ is a one way implication. $\forall h : (h \in I \rightarrow h \in [m_0, m_1])$ but $\neg\forall h : (h \in [m_0, m_1] \rightarrow h \in I)$. As a result, $|I| \leq |[m_0, m_1]|$.*

It is important to understand that m_0 and m_1 play a double role by encoding the extent of the intersection I while also being the minimum and maximum possible values for any $h \in I$. Figure 1 shows an example with a 3-dimensional (hyper)-cube split in two parts, the front with $y = 1$ and the back with $y = 0$. If we assume an I that consists only of the half of the cube that has $y = 1$ while $x, z = 0, 1$ are unconstrained then we would get $m_0 = 010$ for constraining y's lower dimension and $m_1 = 111$ because there are no other constraints. As we can see, m_0 and m_1 represent the numerical minimum and maximum of the intersection I which consists of the front of the cube.

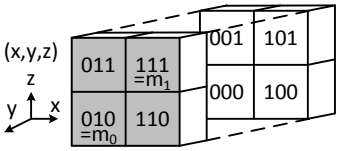


Fig. 1: Hypercube with $k = 3$ where I (grey area) is constrained to $y =$ '1'

Corollary 3 (Size of $N : |N|$)**.** *The size of N can be calculated from m_0 and m_1. Let $n_{1,m0}$ be the number of '1' bits in m_0 and $n_{0,m1}$ be the number of '0' bits in m_1. Then the number of dimensions with restriction $k_r = n_{1,m0} + n_{0,m1}$. Since N extends only in the non-restricted dimensions, N becomes $\mathbb{H}^{k-k_r}$ and the size of N becomes $|N| = 2^{k-k_r}$.*

Since counting '1' bits is a constant time operation on typical modern CPUs, the size $|N|$ can be efficiently computed with (Example in Java):

```
//mask to avoid bits with i>=k
long mask = ~(-1<<k);
//count `1'-bits
int k_r = Long.bitCount(m0 | ((~m1) & mask));
//power of 2^(k-k_r)
int sizeN = 1 << (k - k_r);
```

If we look at the Z-ordered traversal in I, we see that the gaps between h values in I are irregular, see for example the gap between '011' and '110' in Fig. 1. A dimensional restriction on a highly significant bit will cause a big gap while a less significant bit will cause a small gap.

Corollary 4 (Gap size)**.** *We assume an N with a single restriction on one dimension i with* $0 \leq i < k$*. If we want to traverse the gap then we need to move on by* 2^i *entries before we find the next valid entry. The size of the gap, i.e. the number of invalid entries, is* 2^i*. If multiple gaps are crossed at the same time, the total width of the gap is the sum of the individual gaps. The largest possible gap occurs if all restrictions are crossed at the same time. In this case, without further proof, the size of the gap is simply the number resulting from* $(m0 \mid \sim m1) + 1$*.*

5 Algorithm 1: isInI(h,I)

The **isInI()** algorithm checks whether an H-address h intersects with I. To specify the shape of I, we use m_0 and m_1 because they carry all relevant information and uniquely specify I. The function has therefore the parameters **isInI(h,m0,m1)**. We define:

Definition 7. *(Bitstring* b_X *and bits* $b_{X,i}$*) We define b as the bitstring of a given integer number. We use* b_h, b_{m0}, b_{m1} *to indicate that we discuss the bit representation of h,* m_0 *and* m_1 *respectively.* $b_{h,i}, b_{m0,i}, b_{m1,i}$ *is the ith bit of b, starting with* $i = 0$ *for the least significant bit.* $0 \leq i < k$ *irrespective of the number of leading '0' bits.* $b_{\{h,m0,m1\},i}$ *is undefined for* $i < 0$ *and assumed to be '0' for* $i \geq k$*.*

While I is continuous in space, see grey area in Fig. 1, it is not contiguous in terms of the numeric sequence of its h-addresses. Therefore, it is not sufficient to check if $m_0 \leq h \leq m_1$. For example, as shown in the figure, the grey intersection with the query box consists of the non-contiguous sequence $\{010, 011, 110, 111\} = \{2, 3, 6, 7\}$.

A naive algorithm could check for each bit $b_{h,i}$ (with $0 \leq i < k$) in h whether it is compatible with the according bits $b_{m0,i}$ in m_0 and $b_{m1,i}$ in m_1. If $b_{m0,i} = 1$ and $b_{h,i} = 0$ or $b_{m1,i} = 0$ and $b_{h,i} = 1$, then $h \notin I$ otherwise $h \in I$. This approach would require $2 * k$ bit comparisons.

A constant time version is presented in [ZZN14], however with little explanation how or why this works. The algorithm in [ZZN14] is as follows:

```
boolean isInI(long h, long m0, long m1) {
  if ((h | m0) != h) {
    return false;
  }
  if ((h & m1) != h) {
    return false;
  }
  return true;
}
```

List. 1: **isInI(), original version from [ZZN14]**

Lemma 1. ***isInI***(h, m_0, m_1) *= true iff* $h \in I$.

Proof. We limit the proof to one individual bit b_i because ⊕ (XOR), & (AND), | (OR) and ~ (NOT) are bitwise operations that process one bit from each operand without interfering with other bits. Therefore, if the proof works for a single bit, it implicitly works for any number of bits. The first check in Lst. 1 (‘`if (((h | m0) & m1) != h)`’) can only return `false` if $h_i \mid m_{0,i} \neq h_i$. This can only happen if $(h_i = 0) \wedge (m_{0,i} = 1)$, which indicates, see Def. 5, that h refers to the ‘0’ half of the hypercube while only the ‘1’ half is part of I. Accordingly, for the second term `h & m1 != h`, this can only return `false` if $(h_i = 1) \wedge (m_{1,i} = 0)$, which indicates, again see Def. 5, that h refers to the ‘1’ half of the hypercube while only the ‘0’-half is part of I. As a result, **inc()** returns `false` iff $h \notin I$. □

Our optimisation is almost trivial, it uses the fact that $m_{0,i}$ and $m_{1,i}$ can never restrict on the same dimension because $m_0 \& (\sim m_1) \equiv 0$, see Def. 6. Using this fact, the code can be simplified to

```
boolean isInI(long h, long m0, long m1) {
  return ((h | m0) & m1) == h;
}
```

List. 2: **isInI(), final version**

In other words, if m_0 requires any bit to be ‘1’ which is not set in z, or if m_1 requires any bit to be ‘0’ which is ‘1’ in h, then the comparison with the original h will fail and the function reports a mismatch.

Lemma 2. $h|m_0 \& m_1 = h \Leftrightarrow h \in I$

Proof. Building on Lemma 1, Lst. 2 can only behave different from Lst. 1 if m_0 and m_1 would affect the same bit and m_0 would be the inverse of m_1, i.e. if `(h | m0) != h` but `((h | m0) & m1) == h`. However, m_0 and m_1 can never affect the same bit, see Def. 6, hence both algorithms show identical behaviour. □

From the code it is obvious that it executes in constant time as long as all values do not have more bits than a CPU register.

6 Algorithm 2: inc(h)

In cases where I is much smaller, see Corollary 3, than the surrounding binary hypercube, i.e. $|I| \ll 2^k$, checking each entry with **isInI(h,I)** is not very efficient. Instead, it is desirable to have a way of directly generating h-addresses that are part of the intersection. One approach is an algorithm that takes one $h_{in} \in I$ as input and generates the next bigger $h_{out} \in I$, thus allowing to generate a complete set of all $h \in I$. To this end we propose the **inc**(h_{in}, m_0, m_1) algorithm that produces all $h \in I$. The starting value for h_{in} would be m_0, i.e. **inc**$(h_{in} = m_0, m_0, m_1)$.

The problem is that I is not a necessarily contiguous sequence of H-addresses but can have numerous gaps of different length. For example, the I in Fig. 1 is $\{010, 011, 110, 111\} = \{2, 3, 6, 7\}$ which has a gap of two in the middle but no gap between the other elements.

That means, depending on the current h_{in}, **inc()** has to add a different $\delta h_{in,out}$ to get h_{out}. We can see that $\delta h_{in,out}$ is related to m_0 and m_1. If the ith bit (counting i from the lowest bit) of m_0 is 1 ($m_{0,i} = 1$) and the ith bit of h_{in} is '0' after adding '1', then we must add an additional 2^i in order to switch the bit back to an acceptable value. The algorithm for m_1 is similar. Such an algorithm that has to verify each bit obviously executes in $O(2 * k)$ since h, m_0 and m_1 have each k bits. Based on the example in Fig. 1, the transition from $h = 011$ to $h = 110$ would look like this:

```
--> start: h=011
h <- h+1 = 100
--> conflicts with m1, bit i=1
h <- h+(2^1) = 110
--> okay
```

An obvious approach to achieve better runtime than $O(k)$ is to find a way to calculate $\delta h_{in,out}$. While we are not aware of a way to do this, we achieve the desired result by exploiting the CPU's *add* operation. The problem is that, if we add '1' to a bit, a possibly resulting overflow should not necessarily go to the immediate next higher bit, but to the next higher bit that is unrestricted and that can therefore be flipped. The trick is to let the CPU's *add* operation skip over the bits that are restricted by m_0 and m_1. To do this, we set all bits that should be skipped to 1. Any overflow will then be forwarded to the next '0' bit in constant time. Since the input h is already a valid H-address (per definition) the bits restricted in m_0 are already set to '1' and we only need to set the restricted bits from m_1 to '1', because they will all be '0' due to the restriction. This can easily be done with

```
h = h | (~m1); //set filtered bits to `1'
```

If we now increment h by one, all bits on which we have restricted create an overflow, unless the overflow is swallowed by a lower order bit that was '0'.

Now we just have to make sure that we turn h back into a valid value by setting and unsetting the restricted bits. This can be done by

```
    h = (h & m1) | m0; //restore filtered bits
```

The resulting function is:

```
long inc(long h_in, long m0, long m1) {
  long h_out = h_in | (~m1); //pre-mask
  h_out++;                   //increment
  h_out = (h_out & m1) | m0; //post-mask
  return h_out;
}
```

List. 3: **inc(), improved version**

This function produces an ordered sequence of $h \in I$ with $\Theta(1)$ per h. Note that this function returns $h_{out} \leq h$ if the incoming $h_{in} = m_1$. The reason is that in a practical implementation, m_1 should have all bits b_i for $i \geq k$ set to '0', which means that $h_{out} \leq h$.

Corollary 5 (Stop condition)**.** *If the input is $h_{in} = m_1$ then $h_{out} = m_0$, unless the CPU performs signed computation and the overflowing bit causes a sign change to a negative value. Also, in the special case of $m_0 = m_1 \rightarrow h_{out} = h_{in}$ because $h = m_1 \rightarrow h_{out} = m_0 = m_1 = h_{in}$.*

We split up the proof of **inc()** in three parts. First we show that **inc()** wraps around from m_1 to m_0, i.e. that $\mathbf{inc}(m_1, I) = m_0$.

Lemma 3. ***inc***$(h_{in} = m_1, m_0, m_1) = m_0$*. This wrap around condition means that the highest possible values $h_{in} = h_{max} \in I$, which is m_1, results in the lowest possible value, m_0. This condition ensures that $h_{out} =$**inc()** always produces $h_{out} \in I$, however, as we will see, at the cost of breaking the $h_{in} > h_{out}$ rule (Lemma 5).*

Proof. In the first operation all bits are set to '1': $h_{out} = m_1 \mid {\sim}m_1 = \{1\}^k \rightarrow h_{out} = 2^k - 1$. In the second step we add 1, resulting in $h_{out} = 2^k$, which is all '0' with a leading '1' at position $i = k + 1$. In the third step, the $\&m_1$ operation sets the leading bits to '0', resulting in $h_{out} = 0$, and the $h_{out} = h_{out} \mid m_0$ results in $h_{out} = m_0$. □

Now we show that **inc()** always returns an $h \in I$.

Lemma 4. $\forall h_{out} : h_{out} = \mathbf{inc}(h_{in}, m_0, m_1) \rightarrow h_{out} \in I$

Proof. For any H-address $h_{out} = \mathbf{inc}(h_{in}, m_0, m_1) \rightarrow h_{out} \in I$ for all valid m_0, m_1 and $h_{in} \in I$ because the masking $h\&m_1 \mid m_0$ (post-masking) ensures that only the valid bits remain set, i.e. that $h_{out} \in I$. In other words, let $C(h) = h\&m_1 \mid m_0$ be the function used in the the post-masking step in **inc()** and in **isInI()**, see Lemma 1. C is idempotent since '&' and '|' are idempotent bitwise operations. Since **inc()** applies $C(h)$ as the last step, a subsequent check with **isInI()** results in $C(C(h))$. However, $C(C(h)) \equiv C(h)$, hence $\neg\exists h : C(C(h)) \neq C(h)$ which means that $\forall h_{out} : h_{out} = \mathbf{inc}() \rightarrow h_{out} \in I$. □

Finally we show that **inc()** always returns the direct successor, i.e. that there is no valid $h \in I$ that lies between any given input value and output value. This implies that $h_{out} > h_{in}$ for $h_{in} < m_1$.

Lemma 5. $\neg\exists h_x : (h_{in} < h_x < h_{out}) \wedge (h_{out} = \textbf{\textit{inc}}(h_{in}, m_0, m_1))$*, or simply* $h_{in} \prec h_{out}$ *where '*$\prec$*' indicates direct predecessor relationship in I.*

Proof. First we consider the case that the least significant bit $b_{h,0} = 0$ right before the increment (after the pre-masking). $b_{h,0}$ can only be '0' if m_0 does not restrict on this bit, i.e. $m_{0,0} = 0$, otherwise $h_{in} \notin I$ would not be a valid input argument. Neither can m_1 restrict on the first bit, otherwise the initial pre-masking $h \mid {\sim}m_1$ would have set it to '1'. During the increment, we add '1', resulting in a bitstring that represents an integer that is trivially '1' larger than h_{in}. The post-masking has no effect on $b_{in,0}$ because we established that neither m_0 nor m_1 can have a restriction on that bit. They also cannot change any of the other bits of h_{out} because these have not changed and are identical to the bits of h_{in} which comply by definition with all restrictions imposed by m_0 and m_1. Since adding '1' to $b_{h,0}$ cannot affect any other bits, nor is it affected *by* other bits, it can be generalised to adding a '1' bit to any '0' bit, i.e. for any $b_{h,i}$.

Next, we show that adding '1' to any '1' bit $b_{h,i}$ also works. After the pre-masking step, a bit $b_{h,i}$ can be '1' either because it was '1' in h_{in} or because it was set to '1' during pre-masking because m_1 has a restriction on that dimension. Now, adding '1' to a '1' bit in $b_{h,i}$ will cause an overflow and result in $b_{h,i} = 0$ and '1' added to the next higher bit $b_{h,i+1}$. The overflow may cascade through several bits until it adds '1' to a '0' bit in $b_{h,j}$, with $j > i$, which we treated already in the first part of this proof. If neither m_0 nor m_1 impose any restrictions, the post-masking will not change h_{out} and h_{out} is trivially the +1-successor of h_{in}. Since the highest modified bit $b_{h,j}$ was a '0' bit, $h_{out} > h_{in}$ holds as established above. We also know from Lemma 4 that $h_{out} \in I$.

Finally, is h_{out} the direct successor of h_{in} or, in other words, could there be a value h_x with $h_{in} < h_x < h_{out}$? Any $h_x > h_{in}$ must have at least one '1' in a position i_x where h_{in} has a '0', otherwise it cannot be greater than h_{in}. This is only possible if '1' and '0' are actually possible values for that position, which means that neither m_0 nor m_1 impose a restriction on that position. If the position in question is $i_x = 0$, then h_{out} is the immediate successor as shown in the first part of this proof. If $i_x > 0$ then h_{out} is also the successor of h_{in} because, as shown above, the algorithm will overflow until it hits a bit that is '0'. All bits before that (i.e. all that cause an overflow) are either '1' in h_{in} or they can have only one state, which means h_x could not be different from h_{in} at that position without violating the boundary imposed by m_0 and m_1. This means there cannot be an h_x with $h_{in} < h_x < h_{out}$. □

7 Algorithm 3: succ(h,i)

The third algorithm works similar to **inc(h,I)**, but accepts as input arbitrary $h \in \mathbb{H}^k$ and not only $h \in I$. We start with a version of the algorithm that treats three cases separately before we present a more compact but less intuitive version without branching.

The 3-cases implementation first checks whether $h \in I$ which means that **inc()** can be used from incrementation (1st case). If $h \notin I$, it finds out which bits collide with m_0 (2nd case) or m_1 (3rd case) and, depending on where the most significant of the colliding bit comes from, uses two approaches to generate an output $h_{out} \in I$. The details are given in the proof to Lemma 6.

```
long succ(long h, long m0, long m1) {
  if (isInI(h, m0, m1)) {
    return inc(h, m0, m1); //1st case
  }

  long coll = ((h | m0) & m1) ^ h;
  long diffBit = maxBit(coll);
  long mask = diffBit > 0 ? diffBit-1 : 0;

  long confM0 = (~h) & m0;
  long confM1 = h & ~m1;

  if (confM0 > confM1) { // 2nd case
    h &= ~mask;
    h |= m0;
    return h;
  }

  //increment - 3rd case
  long out = h | ~m1;     //pre-masking
  out += coll & ~m1 ;     //increment
  out = (out & m1) | m0; //post-masking
  return out;
}
```

List. 4: Implementation of **succ()**

Note that we use a function **maxBit(x)** that returns a "*value with at most a single one-bit, in the position of the highest-order ("leftmost") one-bit in the specified value 'x'. Returns zero if the specified value has no one-bits in its two's complement binary representation, that is, if it is equal to zero*"[6]. Modern CPUs typically provide a constant time instruction for this operation.

Lemma 6. *h_{out} =**succ**(h_{in}, m_0, m_1) $\rightarrow$ $h_{out} \succ h_{in}$ **with** $h_{out} \in I$. That means **succ()** always returns the next possible $h \in I$, i.e. $\neg\exists h_x : h_x \in I \wedge h_{in} < h_x < h_{out}$.*

Proof. We determine which bits in h_{in} conflict with m_0 or m_1, i.e. in which dimensions h_{in} lies outside I. Let i_{m0} and i_{m1} be the position (the rightmost and least significant bit is at $i = 0$) of the most significant bit $b_{h,i}$ that poses a conflict with m_0 and m_1, respectively, or $i = -1$

[6] See javadoc of `Long.highestOneBit()` in JDK 7 by Oracle Inc.

if no conflict exists. We now consider three scenarios. If $i_{m0} = i_{m1}$ then $i_{m0} = i_{m1} = -1$, because m_0 and m_1 can never conflict on the same bit, see Def. 6. If there is no conflict we can simply apply **inc()** and finish.

As second case we consider $i_{m0} > i_{m1}$. In this case, $b_{i,m0}$ is set to '0' even though it would need to be set to '1' to intersect with I. To create the next highest valid $h \in I$ we simply set all bits $b_i : i \leq i_{m0}$ to their respective minimum, i.e. we set all bits b_i with $i \leq i_{m0}$ to the values of the last i_{m0} bits of the known minimum m_0. The result is the smallest $h \in I$ with $h > h_{in}$, i.e. $h \succ h_{in}$, see also Corollary 2.

As third and last case we consider $i_{m0} < i_{m1}$ where the most significant conflicting bit at i_{m1} has a '1' instead of the required '0'. To find the next higher $h \in I$, we first apply the pre-masking from **inc()**, then we add a '1' at the conflicting position i_{m1}, i.e. we add $2^{i_{m1}}$, then we set all bits b_i with $i \leq i_{m1}$ to their respective minimum defined by m_0 and finally apply the post-masking from **inc()**. By means of an overflow during the addition, the conflicting bit is set to '0' and the addition ensures that the resulting number is larger than h_{in}. By setting the trailing bits to their minimum, we ensure that we do not skip any $h \in I$, i.e. we ensure that $h \succ h_{in}$. To ensure that adding '1' works fine for the higher order bits b_i with $i > i_{m1}$, we apply the pre-masking and post-masking from algorithm **inc()** which ensures that the resulting value is indeed $h \in I$, see Lemma 4. □

While the algorithm in Lst. 4 works fine, it can be optimised by avoiding the three branching statements. Lst. 5 shows an optimised version with less instructions and without branching. First, we calculate two values, `confM0` and `confM1`, that have exactly one bit set to '1', either at the most significant position where a conflict occurs or as the least significant bit. We then calculate two masks, `maskM0` and `maskM1`, that are filled with '0' up to and including the '1' bit of `confM0` and `confM1`. After the '1' bit, they are filled with '1'. If no conflict occurs, the masks are all '0'. Then we start the incrementation with the known pre-masking step. This is especially necessary to bring the most significant conflicting bit into a consistent state. Then, in the new *masking* step, we remove all bits below the most significant conflicting bit. After that, we add `confM0 | ~maskM1`. If no conflict occurs, `confM0` is '1' and we add '1'. Otherwise we add '1' at the most significant conflicting position with m_1, unless m_0 has a more significant conflicting position, in which case the mask `~maskM0` ensures that nothing is added. Finally we do the post-masking and return the result. It is easy to see that the algorithm completes in constant time.

```
long succ(long h, long m0, long m1) {
  long confM0 = maxBit((~h) & m0 | 1);
  long confM1 = maxBit(h & ~m1 | 1);
  long maskM0 = confM0-1;
  long maskM1 = confM1-1;

  //increment
  long out = h | ~m1;       //pre-masking
  out = out & ~(maskM0 | maskM1); //mask
  out += confM1 & ~maskM0; //increment
```

```
    out = (out & m1) | m0;    //post-masking
    return out;
}
```

List. 5: Optimised no-branch **succ()**

8 Application Example

For the remainder of the paper, we use the PH-tree [ZZN14, Zä15] as a running example for the applicability of the established algorithms. The PH-tree is a index that forms a hierarchy of binary hypercubes. The root node is a binary hypercube and each corner is connected with a child hypercube which recursively have hypercubes connected to their corners. The position of the corner of the k-dimensional binary hypercube in the root node encodes the first bit of all values p_i of a point p, this effectively interleaves the bits of all p_i. The first child hypercube encodes the 2nd bit, and so forth. When storing values with a precision of 64 bit, the tree has a depth of 64 nodes. For simplicity we assume that all values p_i have the same precision and we ignore the fact that the trees may, for optimisation, create nodes only if at least two corners have children. Since each hypercube represents one bit of each dimension of a stored point, the tree can be at most 64 nodes deep for 64 bit values.

Geometrically, the PH-tree bipartitions the space recursively into hypercubes of decreasing size which are represented by nodes. Division is limited such that each node contains at least two points, either directly stored in the node or in subnodes. When traversing the tree one node after another, the data points p are returned in Z-order according to the interleaved value $z = \textbf{interleave}(p_0, ..., p_{k-1})$. An example of a two-dimensional tree ($k = 2$) with 3-bit values ($w = 3$) is shown in Fig. 2. The numbers are the interleaved z values, the squares represent the nodes. The outer square is the root node, it contains $2^k = 4$ child nodes, which each contain 4 leaf nodes with 4 points each.

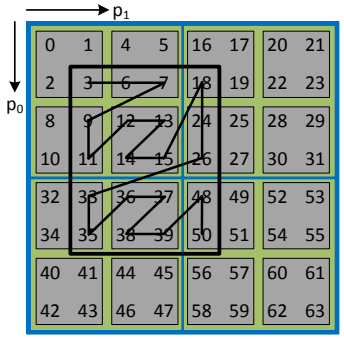


Fig. 2: z values and nodes of a tree with $k = 2$ and $w = 3$ with one root node (blue), its child nodes (green) and 16 leaf nodes (grey). The black rectangle is the querybox intersecting with binary hypercube nodes in z-ordered tree.

The PH-tree supports three storage representations for internally representing the sub-nodes and points stored in a node: Array Hypercube (AHC), List Hypercube (LHC) and Nested Tree Hypercube (NTHC).

In AHC representation, the entries of a node are stored in an array of size 2^k. This representation requires memory in the order of $\Theta(2^k)$, but can be efficient if the nodes contains many entries. The array addresses are effectively the Z-addresses which allows very fast lookup in $\Theta(1)$. The cost of full traversal is $\Theta(2^k)$, insert and delete execute in $\Theta(1)$.

In LHC representation, all entries are stored in a sorted list ordered by their Z-addresses. This approach is often more memory efficient but requires a binary search for random access with $O(\log_2 |N|)$. A full traversal costs $O(|N|)$.

With NTHC, entries are stored in a nested tree, see [Zä15]. NTHC is only used for very large nodes to speed up insertion and deletion. Like LHC, the cost of random access is about $O(\log |N|)$ and full traversal is $O(|N|)$. For the purpose of the calculations below, NTHC behaves approximately like LHC, except for higher base cost for all operations. Hence, we discuss below only LHC and imply that any conclusion also apply to NTHC.

One property of the PH-tree is that all entries in a node share the same prefix, i.e. the leading bits of the Z-values in a node are identical. This is called the *node-prefix*. Figure 3 shows on the left an example with the prefixes for the 4 sub-nodes of a root node with $k = 2$. The prefix is essentially the coordinate in the root's hypercube where the sub-node is attached. On the right the figure shows the prefixes of the leaf nodes, consisting of the prefix of their parent plus their own position in the parent. A prefix always contains a multiple of k bits.

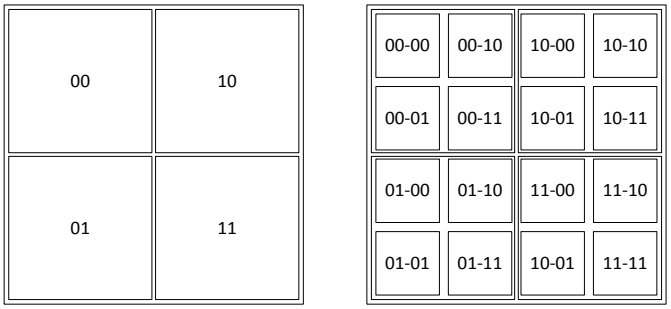


Fig. 3: Prefixes (positions in global space) of the direct children of the root node (left) and their respective children (right)

8.1 Window Queries

Window queries are specified by a 'lower left' and 'upper right' corner q_{min} and q_{max}. To locate matching points inside the query box, the PH-tree traverses the tree by locating the node that contains the interleaved Z-value of q_{min}. The tree then iterates through the stored Z-values of each node. Once it reaches the end of a node it returns to the parent node and traverses the next child node. The algorithm stops once the current Z-value is $\geq q_{max}$. Figure 2 shows an example with a query box with $z_{min} = \textbf{interleave}(001, 001) = 3$ to $z_{max} = \textbf{interleave}(101, 100) = 50$. The algorithms in this paper can be used to efficiently traverse the content of a node so that only those child nodes and points are traversed that potentially intersect with the query box.

For a window query with a query box defined by q_{min} and q_{max}, we intersect the query box iteratively with the binary hypercubes of each node. Starting with the root node, we need to identify those quadrants of the root's hypercube that can intersect with the query box. A simple approach is to iterate through all quadrants (represented by H-adresses), calculate the boundaries of each quadrant and check whether they overlap with the query box. Overlapping occurs if the lower boundary $p_{min} \leq q_{max,i}$ or the upper boundary $p_{max} \geq q_{min,i}$. To translate the boundaries of a hypercube relative into the global space, we need apply the prefix of the node. A naive implementation may look like this:

```
boolean isInI(h, qMin, qMax, prefix) {
  for (0 <= i < k) {
    //convert the i'th bit of h to int
    int pMin = hToMinInt(prefix, h, i);
    int pMax = hToMaxInt(prefix, h, i);
    if (pMax < qMin[i] || pMin > qMax[i]) {
      return false;
    }
  }
  return true;
}
```

List. 6: naive isInI($z, qMin, qMax, prefix$)

The `hToMinInt()` and `hToMaxInt()` in the above algorithm hide the fact that `p` needs to be reconstructed before it can be compared to `qMin` and `qMax`, i.e. we need to construct a minimum and maximum k-dimensional integer `p` consisting of the k-dimensional `prefix` and h. The difference between the two functions is that, if we are not in the leaf level, `hToMinInt()` fills all remaining bits with '0' while `hToMinInt()` fills them with '1'. On the leaf level, there are no remaining bits to be filled. Figure 4 shows an example where a node [8, 11] intersects with a query so that only {9, 11} lie in I. The middle part of the figure shows how the tree encodes the values, with $00-10$ as the node's prefix and the four h-values 00, 01, 10 and 11. The right part shows the same h-values and the resulting m_0 and m_1 which encode the intersection I (grey area) and, at the same time, represent the minimum and maximum h-values in I.

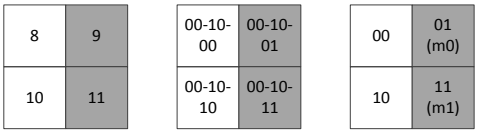


Fig. 4: Intersection I (grey area) of node with a query-box (left), according bit-encoding with prefix $00-10$ (middle) and resulting quadrants with m_0 and m_1 (right)

In the following sections we discuss how and when the three proposed algorithms can be used for window queries.

8.2 isInI()

The **isInI()** algorithm is useful when iterating through the H-addresses of an AHC hypercube where most elements can be expected to lie inside I, i.e. $|I|$ approaches 2^k. Since m_0 and m_1 can serve as minimum and maximum values for the iteration, it is sufficient if a suitable majority of the elements in $[m_0, m_1]$ can be expected to be in I. The total cost C_{node} of traversing a whole node in order to identify all potential matches is $C_{AHC,isInI} = 2^k \times (c_{isInI} + c_{array-lookup})$, where c_{isInI} denotes the cost of a single call to **isInI()**. The main advantage is that **isInI()** is very fast because it contains very few operations.

In the case of LHC, full traversal means traversing a list ($c_{list-next}$ per element) of $|N|$ elements. This results in a total cost $C_{LHC,isInI} = |N| \times (c_{isInI} + c_{list-next})$. Therefore, with LHC, **isInI()** is useful if $|I|$ approaches $|N|$ but inefficient for $|I| \ll |N|$.

8.3 inc()

In the following, $\{z_1, z_2, ...\}$ designates a sequence of z values returned that can be constructed from a node's prefix and the H-addresses generated by **inc()**. $[z_1, z_2]$ is a subnode that potentially contains all z-values between z_1, z_2, assuming that these z values are present in the tree.

Figure 2 shows a tree with a query box Q and the ordering of points in the query result. The aim of **inc()** is to return only h values from a node's intersection I so that a traversal for a window query only checks sub-nodes and points that potentially intersect with I. In the example, on the level of the root node (blue), **inc()** returns all children (green nodes), because they all intersect with Q. On the intermediate level (green nodes) it will return all children (grey leaf nodes) for the first node, the two leaves nodes $\{[16, 19], [24, 27]\}$ for nodes for the second and $\{[32, 35], [36, 39]\}$ for the third node and only the first leaf node $\{[48, 51]\}$ for the last green node. On the leaf level (grey nodes) it returns $\{3\}$, $\{6, 7\}$, $\{9, 11\}$ and $\{12, 13, 14, 15\}$ from the top left quarter of the tree, and so forth for the remaining tree.

In other words, **inc()** returns from a node only z values that potentially intersect with the query box Q. However, it does not guarantee that the nodes actually contain any point $p \in Q$. For example, the algorithm will check the top left grey leaf node $\{[0, 3]\}$ even if 3 is not actually stored in the tree. Please note that while the example in Fig. 2 looks quite simple, the size of the nodes grows with 2^k, which means that simply checking all z in a given node becomes prohibitively expensive for large k. Even checking only all actually existing h and the according z in a node, if such a list is available, can be expensive if the node intersects only with a small part with Q.

Using **inc()** in an AHC node costs $C_{AHC,inc} = |I| \times (c_{inc} + c_{array-lookup})$. When using LHC, **inc()** is more expensive with $C_{LHC,inc} = |I| \times (c_{inc} + c_{list-lookup})$ where $c_{list-lookup}$ is a $O(\log |N|)$ operation for performing a binary search on the sorted list of subnodes and point in the node. Obviously, **inc()** tends to work well if $|I|$ is small compared to the number of entries in the node $|N|$ or compared to the maximum size of the node $|\mathbb{H}^k| = 2^k$. The

disadvantage is that this approach always traverses $|I|$ elements even if $|I|$ is much bigger than number of entries in the node, i.e. if $|I| \gg |N|$.

8.4 succ()

We saw that **isInI()** and **inc()** both have strengths and weaknesses depending on the size of N, I and $\mathbb{H}^k$. We also saw in Corollary 4 that the gaps in the value space of I that occur during traversal of a node can be quite irregular and large. One idea for improvement is therefore to traverse areas with large gaps with **inc()** while using full traversal with **isInI()** on the (mostly) contiguous stretches of I. The distance to the next h can be calculated with $\Delta_h = \mathbf{inc}(h, m_0, m_1) - h$, at least for $h \in I$. If Δ_h is large, then we can decide to simply add Δ_h to the current h and continue traversal there. The problem is that the current h may not be from I, because we reached it via full traversal with **isInI()**. This is where **succ()** can be used, because it works with any $h \in \mathbb{H}^k$ as input. This is especially useful in the case of LHC, where random access is expensive due to the required binary search. The complexity of **succ()** is the same as **inc()**, however it has a higher base cost due to the additional operations.

8.5 Algorithm Selection

A detailed cost analysis is beyond the scope of this paper, however we provide a short guide on how to decide whether full traversal with **isInI()** or traversal of the intersection with **inc()** should be more efficient. In the case of AHC we established two relationships:

$$C_{AHC,isInI} = 2^k \times (c_{isInI} + c_{array-lookup}) \tag{1}$$

$$C_{AHC,inc} = |I| \times (c_{inc} + c_{array-lookup}) \tag{2}$$

The full traversal with **isInI()** is more efficient than using **inc()** if $C_{AHC,isInI} \leq C_{AHC,inc}$. Lets assume that $c_{array-lookup}$ is negligible (ignoring memory access costs) and further estimate that $2 \times c_{isInI} = c_{inc}$ because it has roughly half as many instructions. As a result we see that full iteration should be used if:

$$\begin{aligned} & 2^k \times (c_{isInI} + c_{array-lookup}) && \leq && |I| \times (c_{inc} + c_{array-lookup}) \\ \Rightarrow \quad & 2^k \times c_{isInI} && \leq && |I| \times (2 \times c_{isInI}) \\ \Rightarrow \quad & 2^{k-1} && \leq && |I| \end{aligned} \tag{3}$$

That means we should use **inc()** as soon as at least one dimension is restricted. The size $|N|$ can be calculated as discussed in Corollary 3.

In the case of LHC, **isInI()** should be used if $C_{LHC,isInI} \leq C_{LHC,inc}$. Again, we estimate that $2 \times c_{isInI} = c_{inc}$. We also assume that $c_{list-next}$ is negligible (ignoring memory access

costs) and that $c_{list-lookup}$ costs $(\log |N|) \times (2 \times c_{isInI})$ for the binary search that requires twice as many operations for each step as **isInI()**. This means that full iteration should be used if:

$$\begin{aligned} & |N| \times (c_{isInI} + c_{list-next}) && \leq && |I| \times (c_{inc} + c_{list-lookup}) \\ \Rightarrow \quad & |N| \times c_{isInI} && \leq && |I| \times (2 \times c_{isInI} + (\log |N|) \times (2 \times c_{isInI})) \\ \Rightarrow \quad & |N| && \leq && |I| \times (2 + (\log |N|) \times 2) \\ \Rightarrow \quad & \tfrac{|N|}{2 \times (1 + \log |N|)} && \leq && |I| \end{aligned} \tag{4}$$

For **succ()**, the decision whether full traversal should be used or not can be made not only once per node but for each traversal step. However, this implies the additional cost of calculating the size of the gap and the expected number of possible non-matches. The size of the gap negatively affects the cost of full traversal, however if there are very few entries in the node, then full traversal may still be cheaper because the next element may be beyond the gap, thus allowing faster traversal than with **inc()**. The probability of finding any elements in a gap can be more accurately calculated by also taking minimum and maximum values m_0 and m_1 into account. This can be further refined by considering the number of elements that have already been traversed compared to the spatial fraction of the node that has been traversed, i.e. $n_{found}/|N|$ vs $h/2^k$. At the same time, the complexity of the analysis means that hybrid traversal with **succ()** should probably not be used unless it promises considerable advantages. This results in considerable complexity and, as indicated above, we consider it future work that is outside the scope of this paper.

8.6 Experimental Evaluation

Please note that the purpose of this section is exclusively to confirm the theoretic evaluation. Comparative performance test of the PH-tree with other indexes can be found in [ZZN14]. For the experiments we configured the PH-Tree in a special AHC-only mode. The AHC only mode ensures that the nodes in the tree do not change their representation during insert/update/delete (CUD) operations. This is useful in situations with very frequent CUD operations. Figure 5 shows the result for varying dimensionality of a cube-shaped dataset with 10^5 randomly distributed points between [0.0, 1.0] in every dimension, this is equivalent to the CUBE datasets described in [ZZN14]. The results show that, as expected, the use of **inc()** can increase performance considerably, especially for increasing dimensionality which allows for small $|I|$ compared to $|N|$ in the nodes' hyper cubes.

When allowing LHC mode, AHC nodes become much rarer and the performance gain is much less notable. As discussed in Sect. 8.3, **inc()** is expected to make much less of a difference in LHC nodes because random access in LHC mode requires a binary search with $O(\log n)$ base cost per access. AHC becomes increasingly rare with larger dimensionality because nodes have increasingly rarely enough children to justify AHC mode for optimal

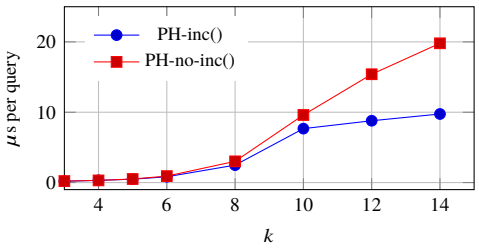


Fig. 5: Query execution times for 10^5 entries with varying dimensionality k of randomly distributed points between [0.0, 1.0] in every dimension. Each query returned on average 1000 entries.

memory usage. As explained [ZZN14], optimal memory consumption is the normally used as decisive factor for AHC vs LHC representation.

We acknowledge that the PH-Tree is not an ideal testbed, and that the non-AHC representation may be rarely used. However, the results clearly demonstrate that the algorithms work and can efficiently avoid worst case cost scenarios.

9 Conclusion

We discussed three algorithms, including two which we developed, that are useful for traversing partitions of k-dimensional binary hypercubes. By exploiting the 'parallel processing' of up to 64 bits in standard CPU registers, all three algorithms execute in constant time for $k < 64$.

For validation we implemented **isInI()** and **inc()** in a specially adapted version of the PH-tree[7]. The algorithms behaved correctly and with the expected performance complexity where our improvements showed increasing effect on the performance with increasing k. This resulted in 20% to 50% reduced query time for $k \geq 10$. Proper evaluation of **succ()** is considered future work.

10 Acknowledgements

This research was partially funded by the Hasler Foundation, Switzerland.

References

[Ba97] Bayer, R.: The universal B-tree for multidimensional indexing: General concepts. In: Intl. Conf. on Worldwide Computing and Its Applications. WWCA '97', pp. 198–209, 1997.

[BCA15] Bøgh, K.S.; Chester, S.; Assent, I.: Work-Efficient Parallel Skyline Computation for the GPU. Proc. of the VLDB Endowment, 8:962–973, 2015.

[7] The PH-tree source code is available from `http://www.phtree.org`

[BKS01] Borzsony, S.; Kossmann, D.; Stocker, K.: The Skyline Operator. In: Proc. 17th Intl. Conf. on Data Engineering. ICDE '01. IEEE, pp. 421–430, 2001.

[Ch15] Chester, S.; Sidlauskas, D.; Assent, I.; Bøgh, K.S.: Scalable Parallelization of Skyline Computation for Multi-Core Processors. In: Proc. 31st IEEE Intl. Conf. on Data Engineering. ICDE '15, 2015.

[Fe02] Fenk, R.: The BUB-tree. In: Proc. of 28th Intl. Conf. on Very Large Data Bases. VLDB '02, 2002.

[KPS91] Kirschenhofer, P.; Prodinger, H.; Szpankowski, W.: Multidimensional Digital Searching and Some New Parameters in Tries. Technical Report CSD TR 91-052, Purdue University, Indiana, USA, 1991.

[LH14] Lee, J.; Hwang, S.-W.: Scalable Skyline Computation Using a Balanced Pivot Selection Technique. Information Systems, 39:1–21, 2014.

[Ma99] Markl, V.: Processing Relational Queries using a Multidimensional Access Technique. Dissertations in Database and Information Systems-Infix, 59, 1999.

[MAK03] Mokbel, M.F.; Aref, W.G.; Kamel, I.: Analysis of Multi-Dimensional Space-Filling Curves. GeoInformatica, 7(3):179–209, 2003.

[Ni13] Nishimura, Shoji; Das, Sudipto; Agrawal, Divyakant; El Abbadi, Amr: \ mathcal {MD}-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services. Distributed and Parallel Databases, 31(2):289–319, 2013.

[OM84] Orenstein, J.A.; Merrett, T.H.: A Class of Data Structures for Associative Searching. In: Proc. of the 3rd SIGACT-SIGMOD Symp. on Principles of Database Systems. PODS '84, pp. 181–190, 1984.

[Ra00] Ramsak, F.; Markl, V.; Fenk, R.; Zirkel, M.; Elhardt, K.; Bayer, R.: Integrating the UB-Tree into a Database System Kernel. In: Proc. of Intl. Conf. on Very Large Data Bases. VLDB '00, pp. 263–272, 2000.

[Sa94] Sagan, H.: Space-filling Curves. Springer, 1994.

[Sa06] Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, 2006.

[Sk06] Skopal, T.; Krátkỳ, M.; Pokornỳ, J.; Snášel, V.: A New Range Query Algorithm for Universal B-trees. Information Systems, 31:489–511, 2006.

[TH81] Tropf, H.; Herzog, H.: Multidimensional Range Search in Dynamically Balanced Trees. Angewandte Informatik, 2:71–77, 1981.

[Zä15] Zäschke, T.: , The PH-Tree Revisited. `http://www.phtree.org`, 2015.

[ZYG11] Zhang, J.; You, S.; Gruenwald, L.: Parallel Quadtree Coding of Large-scale Raster Geospatial Data on GPGPUs. In: Proc. of 19th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems. GIS '11. ACM, pp. 457–460, 2011.

[ZZN14] Zäschke, T.; Zimmerli, C.; Norrie, M.C.: The PH-Tree: A Space-Efficient Storage Structure and Multi-Dimensional Index. In: Proc. of Intl. Conf. on Management of Data. SIGMOD '14, pp. 397–408, 2014.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 485

Optimizing Similarity Search in the M-Tree

Steffen Guhlemann[1], Uwe Petersohn[2], Klaus Meyer-Wegener[3]

Abstract: A topic of growing interest in a wide range of domains is the similarity of data entries. Data sets of genome sequences, text corpora, complex production information, and multimedia content are typically large and unstructured, and it is expensive to compute similarities in them. The only common denominator a data structure for efficient similarity search can rely on are the metric axioms. One such data structure for efficient similarity search in metric spaces is the M-Tree, along with a number of compatible extensions (e.g. Slim-Tree, Bulk Loaded M-Tree, multiway insertion M-Tree, M^2-Tree, etc.). The M-Tree family uses common algorithms for the k-nearest-neighbor and range search. In this paper we present new algorithms for these tasks to considerably improve retrieval performance of all M-Tree-compatible data structures.

Keywords: Metric databases, metric access methods, index structures, multimedia databases, selectivity estimation, similarity search

1 Introduction

Collection and storage of large data sets gives rise to the necessity to also query and process them. The data elements are typically large, unstructured, expensive to process and hardly ever equal to each other. A natural type of query is thus the similarity query.

Hence, there is a need for a general index structure that supports similarity queries on this kind of data. Examples of potential applications are search for genome sequences, fingerprints, and faces [SMZ15], query by example in multimedia databases, machine learning (e.g. k-nearest-neighbor classification or case-based reasoning), etc. Typical query operators are classified in [DD15].

Such an index structure cannot use any structural information, since there is no (common) structure in the data.[4] It can only rely on a generic metric distance function, which must fulfill the axioms of a metric (non negativity, identity of indiscernibles, symmetry, and triangle inequality). Further, this structure has to take into account that a single distance computation can be extremely expensive and that very high dimensional data can fall under the curse of dimensionality. Other requirements are the possibility to store the data on hard disk (i.e. minimize I/O) and to incrementally add or remove entries from the index.

[1] previously TU Dresden, Faculty of Computer Science, Institute for Artificial Intelligence, D-01062 Dresden, Germany, steffenguhlemann@hotmail.com
[2] TU Dresden, Faculty of Computer Science, Institute for Artificial Intelligence, D-01062 Dresden, Germany, Uwe.Petersohn@tu-dresden.de
[3] Friedrich-Alexander-Universität Erlangen-Nürnberg, Faculty of Engineering, Department of Computer Science, Martenstr. 3, D-91058 Erlangen, Germany, klaus.meyer-wegener@fau.de
[4] For some domains like string similarity, special indices using the domain structure exist (e.g. [Fe12] or [Rh10]), but they are not generally applicable.

For similarity search in metric spaces there exists a broad range of structures like BK-Tree [BK73], Fixed Query Tree [Ba94], VP-Tree [Uh91], Bisector Tree [KM83], GNAT [Br95], AESA [Vi86], D-Index [Do03], Metric Index [NBZ11], PPP-Code [NZ14], iDistance [Ja05] and variations of these [CMN01, CMN99, BO97, Yi93, Yi99, DN88, CN00, NVZ92, No93, MOV94, Sh77, BNC03, DGZ03]. A good classification of metric index structures is given by [He09]. Many of these index structures have serious drawbacks, be it for example the restriction to discrete metric spaces (BK-Tree) or a quadratic space complexity (AESA). Further, most of these data structures are inherently static (incremental changes to the stored data are not possible or prohibitively expensive) and only designed to minimize distance computation (and not for example I/O and in-memory tree traversal). An exception is the M-Tree family [CPZ97, Ze06, CP98, Pa99, Sk03, Tr00, Tr02, Ci00] which is thoroughly designed to be a dynamic index structure capable to perform in a broad range of domains and optimizing both I/O and distance computations.[5] This M-Tree family has a compatible structure and shares common query algorithms,[6] which are derived directly from the corresponding queries on a B-Tree with only a few modifications. (Both trees share the basic tree structure.) However, there are subtle differences. For one, the B-Tree divides the space comprehensively and free of overlap, while the M-Tree does not. Second, the B-Tree search only focuses on minimizing the number of disk accesses, while the main goal for the M-Tree is to minimize the number of distance calculations. This leaves room for a more thorough design of the query algorithms with regard to the requirements of similarity search in metric spaces. In this paper new algorithms for the range and the k-nearest-neighbor search are presented. Due to the compatible structure, these new algorithms are applicable to the whole M-Tree family.

The outline of the paper is as follows. In Section 2 the basic structure of the M-Tree family will be presented. Section 3 will introduce some generic optimizations applicable to different kinds of search. The following Sections 4 and 5 will present specific optimizations of the range search and the k-nearest-neighbor search, respectively. Finally Section 6 will show some experimental results.

2 The M-Tree

2.1 The M-Tree Structure

The M-Tree [CPZ97, Ze06] family is used to index similarity in general metric spaces relying only on a metric distance function. The trees grow and shrink dynamically as new data are added or deleted. It is a multi-branch tree that can be configured to minimize I/O. Its basic structure is quite similar to the B-Tree and the R-Tree.

[5] A project seminar [La11] compared different index structures in different domains. The M-Tree had by far the best query performance in terms of necessary distance calculations.

[6] Note, however, that there is a broad range of structures that also pretend to be an extension to the M-Tree, like the Pivoting M-Tree [Sk04], the (B)M^+-Tree [Zh05] or the CM-Tree [AS07], but are not compatible to the M-Tree and also do not have the advantages of the M-Tree. For example the (B)M^+-Tree can only handle Euclidean vector spaces for which better approaches like the kd-Tree exist.

An M-Tree is structured as a hierarchical tree of hypersphere-shaped nodes. Each node entry consists of a routing (pivot) element (which is the center of the hypersphere) and an allowed distance between pivot and data stored below this node (the radius of the hypersphere). Each node can have multiple subnodes up to a predefined capacity limit. All data belonging to a subnode must have a distance to the parent pivot that is smaller than radius of the parent-node hypersphere. Leaf nodes store links to the actual data. The tree grows in a B-Tree-manner bottom-up, i.e., all leafs are at the same level (see Figure 1).

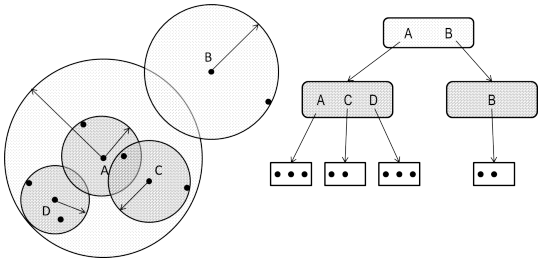


Fig. 1: Structure of an M-Tree [CPZ97]

During insertion of an entry or subnode c into a node n, the distance between n and c must be computed (e.g. to adjust the node radius). As a first optimization, Ciaccia et al. [CPZ97] proposed in their original publication to store the parent-child distances along with the child-node pointers. In search, these precomputed distances are used in conjunction with the triangle inequality to bound the range of the distance of the child pivot to the query object without actually computing it.

2.2 Basic Search Algorithm

Like common tree-search algorithms, similarity search in the M-Tree is basically a hierarchical tree descend pruning nodes whenever possible. In classic main-memory structures the focus is on reducing tree-traversal operations. If data is persisted on paged external memory, a second focus is on reducing I/O operations. For general metric spaces a third optimization goal is necessary: Distance computations have to be avoided as they are usually way more expensive than standard tree-traversal operations, and sometimes even I/O operations.[7]

In general, similarity queries in metric spaces can be rewritten as the search for all data located in a hyperspheric query region centered in a query object. In case of the range query, the radius is fixed as part of the query arguments. In case of a k-nearest-neighbor query, the final search radius has to be determined during search.

The search starts at the root node. It always keeps a queue of unexpanded nodes, which may contain data fulfilling the query. As a node is removed from the queue, its child nodes are

[7] As a simple example consider the Levenshtein edit distance of long texts. A basic distance calculation consumes time and space of $O(N^2)$ where N is the length of the texts. Reading the text from disk will require some time of $O(\texttt{Seek} + \texttt{Read} \cdot N)$ where `Seek` and `Read` are constants. So, if the text is long enough, computing the distance will easily exceed the time to read the text from disk. The problem is worse for most multimedia domains, where one similarity calculation can be a complex optimization on its own.

examined. If it can be guaranteed that the child node cannot contain search results, it can be pruned. Otherwise it is enqueued in the expansion queue. If a leaf node is removed from the queue, the actual data in it are classified as fulfilling the query criteria or not.

To prune a node n it has to be guaranteed that its hypersphere (center n, radius r_n) does not intersect the query hypersphere (center q, radius r_q).[8] In other words, it must be proven that no data element e below n is closer to q than r_q. In the following this closest possible distance is denoted by $d_n^\perp$ – implicitly referencing the current search center. For an inner node this minimal possible distance is calculated using an (expensive) distance calculation $d_{n,q}$ and the triangle inequality by $d_n^\perp = \max(0, d_{n,q} - r_n)$. On the leaf level, the formula is simplified to $d_e^\perp = d_{e,q}$. A node (or leaf) is pruned, if $d_n^\perp > r_q$.

3 General Search Optimizations

The search approach described in Section 2.2 can be improved in a number of aspects independent of the actual type of search.

3.1 Generalization of Existing Optimizations

During examination of a node n the minimum possible element distance $d_n^\perp$ needs to be calculated to make a justified pruning decision. By default, this calculation involves a distance computation which is the search-cost driver.

A basic idea is to use heuristics to inexpensively retrieve a lower bound $\perp_n$ on the distance $d_{n,q}$. Based on $\perp_n$ a lower bound $d_{n,relaxed}^\perp$ on $d_n^\perp$ can be found without any distance calculation:

$$d_{n,relaxed}^\perp = \max(0, \perp_n - r_n).$$

In certain situations, it is possible to prune n based on $d_{n,relaxed}^\perp$ only, saving the distance calculation $d_{n,q}$ entirely. It has to be observed that $\perp_n$ must never overestimate $d_{n,q}$. Otherwise a node might be pruned which contains valid results, leading to an incorrect search. On the other hand, $\perp_n$ should be as close as possible to $d_{n,q}$ as this allows to prune nodes more often. A further criterion for an efficient search is that the effort of the calculation of $\perp_n$ has to be negligible compared to the computation of $d_{n,q}$. Otherwise only distance computations are saved, but search time is not.

In the literature several examples of such tests exist. Unfortunately they are hard-coded into the respective algorithm without considering generalizability.

[8] Note that in case of a k-nearest-neighbor query the query radius r_q will only be determined during the search. However, there will always be a known upper bound on this radius.

3.1.1 Precomputed Distance to Parent Node

One of the optimizations was proposed by Ciaccia et al. [CPZ97] as part of the classic M-Tree. As described in Section 2.1 each M-Tree node stores the distance to its direct parent node. This neither increases the insert effort[9] nor the storage complexity[10]. Since during search a child node c is only expanded after it has been stated that its parent node p cannot be pruned, the distance $d_{p,q}$ has already been computed at the time of the decision on c. Using the precomputed parent-child distance $d_{c,p}$ stored in c, a quick calculation of a bound for the distance $d_{c,q}$ is possible without actually computing it:

$$d_{c,q} \geq \perp_{n,ParentDist} := |d_{p,q} - d_{c,p}|.$$

3.1.2 AESA Principle

Aronovich and Spiegler [AS07] proposed something as part of the CM-Tree, which can be generalized as node-local AESA [Vi86] principle.[11] Inside each node all bilateral distances between child nodes are stored. If some child node c_i of a parent node is examined (involving the distance computation $d_{c_i,q}$), the distance to other child nodes c_k can be bound using this distance and the precomputed distances d_{c_i,c_k}. The lower bound of the distance to c_k is the maximum of all bounds based on already computed distances to c_i:

$$d_{c_k,q} \geq \perp_{n,AESA} := \max_i \left(d_{c_i,q} - d_{c_i,c_k} \right).$$

Contrary to the basic optimization of [CPZ97] (Section 3.1.1), insert effort and storage complexity (in terms of node capacity) are increased. Further, this tree is not fully compatible to the classic M-Tree, i.e., it needs different insert, delete, and query algorithms.

3.1.3 Domain-specific Heuristics

Bartolini et al. [BCP02] proposed the use of a specific M-Tree structure for indexing Levenshtein edit distances. They developed specific "bag" heuristics to compute a cheaper bound on the edit distance sorting letters into bags.

Their idea can be generalized to allow the M-Tree to make use of domain-specific distance bounds in case such bounds exist. However, as this example shows, the idea must be used with care. The Levenshtein edit distance can be calculated in $O(M \cdot N)$ in terms of the lengths M and N of the two texts. The bag heuristics reduce this time to $O(N + M)$. The

[9] To decide on the parent node to insert a child and to adjust the parent-node radius, this distance must be computed in any case during insert.

[10] Each node stores only one distance. Thus, the storage complexity is $O(1)$. Even in absolute terms this one number is usually negligible as typical metric data (e.g. texts, images, genome sequences) are big. Of course in edge cases (2D-data) this can mean an increase of 33%.

[11] AESA [Vi86] is a similarity index structure that precalculates all bilateral distances. During search these distances can be used to prune objects based on few computed distances and the triangle inequality.

problem here is that search algorithms assume heuristics computations to be nearly for free compared to a single distance computation. That is clearly not the case here – the effects depend on the two string lengths. A better approach would be to use heuristics that can be computed fast in $O(1)$, even if its bounds are not so tight.

3.2 Generalized Use of Bounds

3.2.1 Combination of Bounds

As stated in Section 3.1 multiple domain-specific and domain-independent distance bounds $\perp_n^i$ exist, which have 3 main properties:

- Each $\perp_n^i$ must not overestimate $d_{n,q}$, i.e., $\forall i, n : \perp_n^i \leq d_{n,q}$.
- The $\perp_n^i$ have different precision $Pr_i := d_{n,q} - \perp_n^i$. The smaller Pr_i is, the more distance calculations these heuristics may avoid.
- Each $\perp_n^i$ demands a different computation time T_i. In order to be of any use, T_i must be negligible compared to the time of an actual distance calculation.

Using these properties multiple distance bounds can be combined to a stronger one:

$$\perp_n^{combined} = \max_i \perp_n^i$$

$\perp_n^{combined}$ never overestimates $d_{n,q}$ and has the same or a better precision than any $\perp_n^i$.

This optimization can be applied in a generic manner – i.e., not hard-wiring any specific heuristics into the query algorithm. Domain-specific algorithms can be injected into the index structure either at run time using abstract interface classes describing the properties of a certain metric space, or at compile time using languages like C++ and generic metric-space traits. Each search algorithm would then combine all available heuristics to avoid distance calculations.

Care has to be taken to not replace an expensive distance calculation by a comparably expensive heuristics. As an example consider the optimization described in Section 3.1.3. It has a reduced time complexity of $O(N + M)$ (distance calculation: $O(N \cdot M)$). However, compared to a single in-memory tree traversal or really cheap heuristics (e.g. parent distance - see Section 3.1.1) it is not negligible.[12] Hence, it might be inappropriate to use this optimization for certain data distributions, because it may increase the search time. Further, even if it is used, the algorithm should not just compute the maximum of all bounds, ignoring their different computation times. Instead it should try to subsequently prune a node based on single heuristics (cheap heuristics first). [13]

[12] For the search algorithm it is usually assumed that all tree traversal and heuristics operations are essentially for free compared to a single distance computation. So they are used in abundance. This of course depends on the relative cost of a distance calculation, so it might be valid for typical metric domains (texts, multimedia) but is invalid when for example a low to medium dimensional euclidian distance is used.

[13] [ZS15] propose a cost-benefit ratio to choose between heuristics in the context of multi-feature similarity search.

3.2.2 Upper Bounds

Beside lower bounds on the distance also upper bounds can be computed efficiently and used during search. For example, in case of the parent distance (Section 3.1.1) such upper bound can be computed as

$$d_{n,q} \leq \top_{n,ParentDist} := d_{p,q} + d_{n,p}.$$

Based on this bound, the maximum possible distance $d_n^\top$ to an element e below n can be estimated:

$$d_{e,q} \leq d_n^\top = \top_n + r_n.$$

Such bound $d_n^\top$ can be used to save distance calculations as shown in Section 4.

3.2.3 Domain-specific Text-length Heuristics

In the case of the Levenshtein edit distance we propose the use of text-length heuristics for efficiently computing both a lower and an upper bound. These heuristics are applicable in case all edit operations (insert, delete, replace) are equally weighted.

The lower-bound heuristics rely on the observation that even if for two texts of different length the shorter one is an exact substring of the longer one, the length difference must be edited by either insert or delete operations (depending on which text is longer):

$$d_{n,q} \geq \bot_{n,Length} := ||length(n) - length(q)||.$$

For the upper bound we observe that in the worst case the texts are different in each character. In that case, first all characters of the smaller length of the two texts can be replaced and the remaining characters can be either inserted or deleted (depending on which text is longer). Either way the unweighted edit distance can never be greater than the length of the longer text:

$$d_{n,q} \leq \top_{n,Length} := \max(length(n), length(q)).$$

4 Optimization of the Range-search Algorithm

A range query is the most basic similarity query explicitly specifying the query hypersphere H_Q by its center q and its query radius r_q. Due to this, the query can traverse the tree in any order (breadth first, depth first, ...) where depth first has the lowest space requirement. Independent of the expansion order for each node n it must be decided whether and how to process its child nodes. As a first attempt, in `processHeuristics` (Listing 1), the algorithm tries to process the node based only on the heuristics. If this is not possible, the algorithm falls back to computing the distance $d_{n,q}$ and continues with `processDistance` (Listing 2). In this algorithm the following optimization aspects can be applied independently.

Algorithm 1 Basic Range Query: `processHeuristics`

```
processHeuristics(node n, ⊥_n, ⊤_n):
 d^⊥_{n,relaxed} = max(0, ⊥_n − r_n)

 // either determine, that no overlap is
 // possible (do nothing, return true)
 // or leave the work to
 // rangeQuery/processDistance
 // (return false)
 return d^⊥_{n,relaxed} > r_q
```

Algorithm 2 Basic Range Query: `processDistance`

```
processDistance(node n, d_{n,q}):
 if d_{n,q} − r_n ≤ r_q:
  // overlap
  if n is leaf:
   add n to result set

  else:
   for each child c in n:
    rangeQuery(c)
```

4.1 One-child Cut

A tree of enforced equal leaf depth without balanced node splitting (like the M-Tree) sometimes tends to build "aerial roots" – i.e., chains of nodes with only one child. The efficiency of the search in such a tree is determined by the possibility to prune many subnodes at once when examining a single parent node. This advantage is gone in case a node n has just one child c. The algorithm would make some effort in examining n to determine whether it should spend even more effort to examine the child nodes. In case of only one child c it is better to examine c directly. However, care must be taken in case the centers of n and c differ, as this has effect on some heuristics on the grand children (like the parent-distance heuristics). As shown in Listing 3 we replace the examination of a node n with only one child directly by an examination of its child c without ever computing the distance to n.

4.2 Intelligent Combination of Heuristics

As discussed in Section 3.2.1 several domain-dependent and -independent heuristics can be applicable. They are grouped by their approximate run time. For example, the parent-distance heuristics and the text-length heuristics are considered to be fast, while the bag heuristics

Algorithm 3 Range Query: One-child Cut

```
rangeQuery(node n):
 if n has exactly 1 child:
  set c = the child of n
  rangeQuery(c)

 else:
  ...
```

are considered slow. This grouping and the decision which heuristics to apply must be done by the user when initializing the M-Tree.

When examining a node, first the maximum of all fast lower-bound heuristics and the minimum of all fast upper-bound heuristics are computed. Using these bounds, the algorithm tries to process the node n. (For a discussion of the use of upper bounds see the subsequent sections.) If this is not possible, it tries the processing based on the slower heuristics (taking also into account the bounds computed using faster heuristics). Finally the algorithm (Listing 4) falls back to actually computing the distance.

Algorithm 4 Range Query: Combination of Heuristics

```
rangeQuery(node n):
 for each group in heuristicsGroups:
  ⊥_n^group = max(⊥_{n,i} in group, ⊥_n^{group-1})
  ⊤_n^group = min(⊤_{n,i} in group, ⊤_n^{group-1})

  if processHeuristics(n, ⊥_n^group, ⊤_n^group):
   return

 compute d_{n,q}
 processDistance(n, d_{n,q})
```

4.3 Zero Interval

The easiest optimization (Listing 5) can be applied in case $\perp_n = \top_n$. If the upper and lower bound happen to be the same, we immediately know the actual distance: $d_{n,q} = \top_n (= \perp_n)$. This can be used to process the node as if the actual distance had been computed (i.e. in the best possible manner) without actually computing it.

4.4 Upper-bound Enclosure

Using the upper distance bound, we sometimes come across a situation, where

$$d_n^{\top} = \top_n + r_n \leq r_q.$$

Algorithm 5 Range Query: Zero Interval

```
rangeQuery(node n):
 ⊥_n = ...
 ⊤_n = ...

 if ⊥_n == ⊤_n:
  d_{n,q} = ⊥_n
  processDistance(n, d_{n,q})
  return

 if processHeuristics(n, ⊥_n, ⊤_n):
   return

 compute d_{n,q}
 processDistance(n, d_{n,q})
```

This means that even the furthest possible element below n is inside the query hypersphere. As shown in Listing 6, we therefore add the whole subtree below n to the result set without any further distance calculation to a predecessor (or other node examinations).

Algorithm 6 Range Query: Upper-bound Enclosure

```
processHeuristics(node n, ⊥_n, ⊤_n):
 d_n^⊤ = ⊤_n + r_n

 if d_n^⊤ ≤ r_q:
  add all data elements below n
      to result set
  return true

 else:
  ...
```

4.5 Upper-bound Intersection

The basic range-query algorithm tests for each examined node whether the node and the query hypersphere intersect. If so, the node is expanded, i.e., all direct child nodes are examined.

The basic algorithm uses lower bounds to test whether an intersection of node and query hypersphere is impossible. If the node cannot be pruned based on this information, the actual distance is computed. However, sometimes it is possible to preclude an intersection only using the upper bound without computing the actual distance. Node and query hypersphere

definitely intersect, if

$$\top_n + r_n > r_q \geq \top_n - r_n.$$

This explicitly excludes the possibility of $r_q \geq \top_n + r_n$, which is better handled using the optimization in Section 4.4. In this case, the node n can be expanded (i.e. its child nodes are examined) without computing the distance to n. Listing 7 shows the principle.

Algorithm 7 Range Query: Upper-bound Intersection

```
processHeuristics(node n,⊥n,⊤n):
 // ... first test Upper Bound Enclosure

 //now Upper Bound Intersection:
 if ⊤n + rn > rq ≥ ⊤n − rn:
  for each child c in n:
   rangeQuery(c)

  return true

 else:
  ...
```

This optimization should only be applied with care as it has some negative side effects. The parent-distance heuristics require the distance $d_{n,q}$ to the parent node n to be known when the child node c is examined. If this optimization is used, $d_{n,q}$ is not known any more. Instead we only know a lower ($\perp_n$) and upper ($\top_n$) bound for the distance to the parent node when examining the child nodes. $\perp_n$ and $\top_n$ can still be used to bound $d_{c,q}$, but the child bounds will be more loose, allowing less often to prune without computing distances. So in the worst case the optimization saves one distance computation ($d_{n,q}$) but triggers N other distance computations $d_{c_i,q}$ to all N child nodes. This negative effect can be reduced in case there are other heuristics which compensate for a less tight parent-distance heuristics. For this reason, we apply this optimization only, if other fast heuristics are available.

5 Optimization of the (k-)Nearest-Neighbor Search Algorithm

5.1 Overview

In the k-nearest-neighbor search the k closest elements to a query object q are to be found. There exists an equivalent range query whose range r_q is the distance to the furthest of the k results. (Assuming all distances – especially the kth and $k + 1$th – are not similar.) The problem is that r_q is not known in advance. However, it can be bound in the course of the search. In case k or more nodes are already examined, r_q will never be greater than the kth furthest of all examined elements. Thus, there is always a bound $\perp_{r_q}$ on r_q which shrinks during the search process. Contrary to a range query, results that lie inside a hypersphere $H_Q(q, \perp_{r_q})$ are only result candidates and not final search results. Given an oracle, which

would tell the final r_q, the theoretical minimum of necessary distance computations is that of the equivalent range query, because at least the tree has to be descended using this r_q.

In case of a range query, the order of node expansion and the timing of heuristics usage do not matter. Due to the fixed query hypersphere, node examinations are independent of each other and always deliver the same result. In case of a k-nearest-neighbor query this is not the case any more. A distant node, which would not be expanded in case of an already shrunken radius, may be expanded if examined early in the search (still large radius). On the other hand, expanding many close nodes early in the search course can shrink $\perp_{r_q}$ faster than first expanding distant nodes. Hence, instead of the arbitrary order of range-query tree traversal, the k-nearest-neighbor search should examine promising nodes (with high proximity to q) first. Inverse node proximity P is the closest possible distance that a data element e below n can have to the query center q according to current knowledge. In its simplest version this boils down to $P = \max(d_{n,q} - r_n, 0)$.

5.2 Classic Algorithm

The classic algorithm of Ciaccia et al. [CPZ97] (Listing 8) uses a priority queue to expand nodes in order of their proximity. It makes, however, rather ineffective use of distance bounds. Only before inserting a node n into the priority queue the bound is used to check if n may contain elements closer than the currently known closest k results. This algorithm

Algorithm 8 Basic kNN Search

```
kNN(root):
 compute d_root,q and P_root
 add root to priority queue based on P_root

 while priority queue not empty:
  remove front node n from priority queue
  update ⊥_rq based on n

  if n is leaf:
   add n to result candidates

  else:
   for each child c of n:
    if d_c^⊥ ≤ ⊥_rq:
     compute d_c,q and P_c
     add c to priority queue based on P_c
```

minimizes the number of node expansions, but not the number of distance calculations. The problem is that at the time the heuristic check is made to avoid the distance calculation $d_{c,q}$, the query radius limit $\perp_{r_q}$ is still large. It is only reduced significantly when actual data elements are added to the queue. Due to the sorting of the queue in terms of the proximity of the nodes, at this point only not-so-promising nodes are still in the queue and the search

is almost finished. Another problem is the permanent updating of $\perp_{r_q}$ which is only easy in case of $k = 1$.

5.3 Delayed Distance Computation

We present a better algorithm, which avoids computing $\perp_{r_q}$ completely and minimizes the number of distance computations to the theoretical minimum. The main idea is to delay the actual distance computation. For a node n, first an optimistic proximity $P_n^{\perp}$ is computed based on the lower bound $\perp_n$:

$$P_n^{\perp} = \max(\perp_n - r_n, 0).$$

The node is inserted into the priority queue based on $P_n^{\perp}$. When it is retrieved from the queue, it is the most promising node currently known. At this time the distance calculation can no longer be delayed, as we need to find out how promising the node really is. For this, the actual distance $d_{n,q}$ and the actual proximity P_n are computed. From this, there are two possible continuations:

1. n can be reinserted into the queue based on P_n to expand it (insert its children c based on $P_c^{\perp}$) when it is extracted again.
2. The other possibility is to expand it directly. In this case, the children c of n would be inserted directly into the priority queue, based on $P_c^{\perp}$.

In both cases we do not need extra distance calculations in this step, as the expansion and insertion of the children only needs some tree traversal and heuristics calculation. However, there is a downside of case 2. If expanding the children early,

- the expansion effort could be wasted, as search might be over before n must be expanded, and
- we permanently operate on a longer priority queue during search.[14]

In both cases the algorithm stops if

- the priority queue is empty or
- k elements are found and P of the currently extracted node n is greater than the worst distance of the k elements. n cannot improve the result and the remaining nodes in the queue are known to be worse. (At this point the classic algorithm would continue emptying the queue and only by the parent-distance heuristics removing child nodes.)

It can be shown [Gu16] that both variants lead to the theoretically possible minimum number of distance computations. I.e., the same distances are computed as for an equivalent range query. Listings 9 (Case 1) and 10 (Case 2) show the principle of the two cases.

[14] The effort of a single operation on such a queue is $O(\log N)$ – so all operations will be a bit more expensive.

Algorithm 9 kNN: Delayed Distance Computation (Case 1)

```
kNN(root):
 compute d_root,q and P_root
 add root to priority queue based on P_root

 while priority queue not empty:
  remove front node n from priority queue

  if n was stored based on P_n^⊥:
   compute d_n,q and P_n
   reinsert n based on P_n

  else if n is leaf:
   add n to result elements using d_n,q
   if k result candidates are known:
    return result candidates //we are finished

  else: // inner node
   for each child c of n:
    estimate ⊥_c and P_c^⊥ (using d_n,q)
    insert c based on P_c^⊥
```

5.4 Slim Radii

In [Gu16] we present another M-Tree optimization, where nodes can be shrunken to not just the enclosure of their child nodes, but to only all the data elements below. In this case it is possible that a child node n seems to have a closer proximity than its parent node p.

$$P_p > \tilde{P_n} = \max(d_{n,q} - r_q, 0).$$

Since P_n measures the closest possible distance an element below n can have to q, and all elements below n are also below p, none of them can be closer than P_p. So the maximum of all ancestor P's should be used as P_n.

5.5 Reuse of Range-query Optimization

Most of the optimizations found for range queries (see Section 4) can and should be applied to k nearest neighbors as well. For example, multiple heuristics should be combined, taking into account the computation time of the heuristics. If a node has only one child, instead of the node its child can be put on the queue directly. Also, if upper and lower bound happen to be the same, the distance can be concluded without computing it (zero interval).

Algorithm 10 kNN: Delayed Distance Computation (Case 2)

```
kNN(root):
 add root to priority queue

 set r̃_q = ∞
 while priority queue not empty:
  remove front node n from priority queue

  if P_n^⊥ ≥ r̃_q:
   // no improvement possible as all
   // elements of queue are worse
   return current candidates as result

  compute d_{n,q} and P_n
  if P_n < r̃_q:
   if n is leaf:
    add n to result candidates
    update r̃_q based on d_{n,q}

   else:
    for each child c of n:
     estimate ⊥_c and P_c^⊥ (using d_{n,q})
     insert c based on P_c^⊥
```

6 Experimental Results

6.1 Experimental Design

For evaluating the presented optimizations of the query algorithms, we filled a set of 100 trees (in main memory) with 10'000 elements of representative random data in different domains and queried each tree with 40'000 random queries. As domains we used:

- a range of Euclidean vector spaces (from 2-dimensional up to 15-dimensional). Those data are drawn randomly from a clustered distribution. There are 10 random cluster centers. Data within a cluster follow a gaussian normal distribution.
- the Levenshtein edit distance on sample texts. Those are drawn randomly out of a pool of 270'000 lines of programming source code.
- a similarity function on wafer-deformation patterns in the semiconductor industry. The data are taken from the lithographic step in processing wafers, where deformations have to be corrected during exposure of the wafer. As distance we use the integral of the absolute distance function over the wafer surface. The distance at one point on the wafer surface is the euclidian distance of the two deformation vectors[15].

[15] At each point the wafer has a deviation in x- and y-direction from its nominal position.

Over all trees and queries we counted the number of necessary distance calculations and computed an average per search. Since the resulting query effort varies considerably among the domains (due to the curse of dimensionality), we normalized the resulting effort in relation to the effort of a standard strategy, so that the result visualization becomes more readable.

6.2 Range Search

For range queries we compared 3 algorithms:

1. The naive algorithm without use of distance bounds (“None”),
2. The algorithm presented in [CPZ97] (“classic M-Tree”) which uses distance heuristics based on the precomputed distance to the parent node (see Section 3.1.1) and
3. The combination of all new optimizations (see Section 4, “EM-Tree”).

As shown in Figure 2, over all domains the parent-distance heuristics were able to save a slight amount of distance calculations. The combination of the new optimizations presented

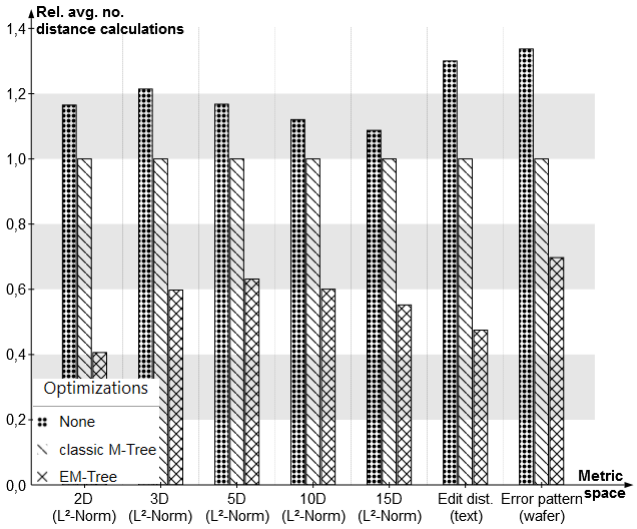


Fig. 2: Experimental Results: Range query

in this paper was able to further reduce the number of distance calculations per search by approximately 40% over all domains. The new algorithm is able to reduce the average range-query search effort considerably.

6.3 (k-) Nearest-Neighbor Search

For k-nearest-neighbor queries again 3 strategies were compared:

1. The naive algorithm without the use of distance bounds (“None”),
2. The algorithm presented in [CPZ97] (“classic M-Tree”) which uses distance heuristics based on the precomputed distance to the parent node (see Section 3.1.1) and
3. The combination of all new optimizations (see Section 4, “EM-Tree”).

As shown in Figure 3, over all domains the classic algorithm presented in [CPZ97] is only able to reduce the number of distance calculations a bit, because the heuristics are used in a quite inefficient manner. Using the new algorithm presented in Section 5 the number of

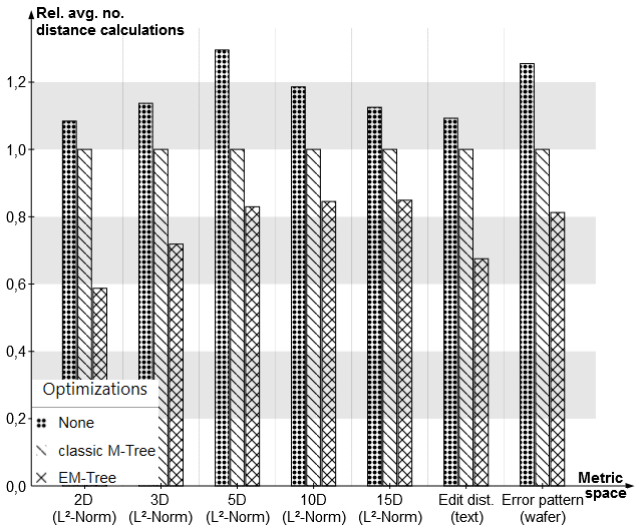


Fig. 3: Experimental Results: kNN query

necessary distance calculations can be further reduced significantly.

7 Summary

In this paper we present new algorithms for a more efficient similarity search on the classic M-Tree [Ze06]. We identified optimization concepts to reduce the number of distance calculations which are the major part of search time (in most metric spaces even outweighing disk-access time). We then applied this approach to develop more efficient algorithms for range and k-nearest-neighbor search. The optimizations are presented in a modular fashion, so that they can be applied independently. In an experimental evaluation with a broad range of metric spaces we were able to show that these optimizations can significantly reduce the number of distance calculations in both query types.

We only implemented the tree in a prototypical manner in main memory. (This is sufficient due to the simple counting for our main goal of minimizing the number of distance calculations.) Future work would be to implement the tree in an efficient manner for use on paginated disks to really compare timings and disk accesses. Also more domains and index

structures should be included in the comparison as in [La11]. Search effort depends on both the algorithms (optimized in this paper) and the tree structure. Contrary to the B-Tree, the M-Tree has degrees of freedom in its structure. We intend to explore the possibility of an optimized use of these degrees of freedom in a manner that the tree can be searched more efficiently. Further, some parameters of the search strongly depend on the domain in their availability, effort and efficiency (e.g. the order and use of different heuristics or the efficiency of single optimizations of the range query). Future work should try to explore the possibility to automatically optimize the algorithmic search (and also tree edit) parameters in accordance with the presented metric space.

References

[AS07] Aronovich, Lior; Spiegler, Israel: CM-tree: A dynamic clustered index for similarity search in metric databases. Data & Knowledge Engineering, 63(3):919–946, 2007.

[Ba94] Baeza-Yates, Ricardo; Cunto, Walter; Manber, Udi; Wu, Sun: Proximity matching using fixed-queries trees. In (Crochemore, Maxime; Gusfield, Dan, eds): Combinatorial Pattern Matching, volume 807 of Lecture Notes in Computer Science, pp. 198–212. Springer, Berlin Heidelberg, 1994.

[BCP02] Bartolini, Ilaria; Ciaccia, Paolo; Patella, Marco: String matching with metric trees using an approximate distance. In: String Processing and Information Retrieval. Springer, pp. 423–431, 2002.

[BK73] Burkhard, W. A.; Keller, R. M.: Some approaches to best-match file searching. Commun. ACM, 16(4):230–236, 1973.

[BNC03] Bustos, B.; Navarro, G.; Chávez, E.: Pivot selection techniques for proximity searching in metric spaces. Pattern Recognition Letters, 24(14):2357–2366, 2003.

[BO97] Bozkaya, T.; Ozsoyoglu, M.: Distance-based indexing for high-dimensional metric spaces. ACM SIGMOD Record, 26(2):357–368, 1997.

[Br95] Brin, S.: Near neighbor search in large metric spaces. In: Proc. Int. Conf. on Very Large Data Bases (VLDB). IEEE, pp. 574–584, 1995.

[Ci00] Ciaccia, P. and Patella, M.: The M2-tree: Processing Complex Multi-Feature Queries with Just One Index. In: DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries. 2000.

[CMN99] Chávez, E.; Marroquín, J.L.; Navarro, G.: Overcoming the curse of dimensionality. In: European Workshop on Content-based Multimedia Indexing (CBMI 99). pp. 57–64, 1999.

[CMN01] Chávez, E.; Marroquín, J.L.; Navarro, G.: Fixed queries array: A fast and economical data structure for proximity searching. Multimedia Tools and Applications, 14(2):113–135, 2001.

[CN00] Chávez, E.; Navarro, G.: An effective clustering algorithm to index high dimensional metric spaces. In: Proc. 7th Int. Symp. on String Processing and Information Retrieval. pp. 75–86, 2000.

[CP98] Ciaccia, P.; Patella, M.: Bulk loading the M-tree. In: Proc. 9th Australasian Database Conf. (ADC). pp. 15–26, 1998.

[CPZ97] Ciaccia, P.; Patella, M.; Zezula, P.: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In: Proc. Int. Conf. on Very Large Data Bases (VLDB). pp. 426–435, 1997.

[DD15] Deepak, P; Deshpande, Prasad M: Operators for Similarity Search: Semantics, Techniques and Usage Scenarios. Springer, 2015.

[DGZ03] Dohnal, V.; Gennaro, C.; Zezula, P.: Similarity join in metric spaces using eD-index. In: Proc. Int. Conf. on Database and Expert Systems Applications (DEXA). Springer, pp. 484–493, 2003.

[DN88] Dehne, F.; Noltemeier, H.: Voronoi Trees and Clustering Problems. In (Ferrate, Gabriel and Pavlidis, Theo and Sanfeliu, Alberto and Bunke, Horst, ed.): Syntactic and Structural Pattern Recognition, volume 45 of NATO ASI Series, pp. 185–194. Springer Berlin Heidelberg, 1988.

[Do03] Dohnal, V.; Gennaro, C.; Savino, P.; Zezula, P.: D-Index: Distance Searching Index for Metric Data Sets. Multimedia Tools and Applications, 21(1):9–33, 2003.

[Fe12] Fenz, Dandy; Lange, Dustin; Rheinländer, Astrid; Naumann, Felix; Leser, Ulf: Efficient Similarity Search in Very Large String Sets. In: Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM). Chania, Crete, Greece, 2012.

[Gu16] Guhlemann, Steffen: Neue Indexverfahren für die Ähnlichkeitssuche in metrischen Räumen über großen Datenmengen. PhD thesis, TU Dresden, April 2016.

[He09] Hetland, Magnus Lie: The Basic Principles of Metric Indexing. In (Coello, Carlos Artemio Coello; Dehuri, Satchidananda; Ghosh, Susmita, eds): Swarm Intelligence for Multi-objective Problems in Data Mining. Springer, Berlin, Heidelberg, pp. 199–232, 2009.

[Ja05] Jagadish, Hosagrahar V; Ooi, Beng Chin; Tan, Kian-Lee; Yu, Cui; Zhang, Rui: iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. ACM Transactions on Database Systems (TODS), 30(2):364–397, 2005.

[KM83] Kalantari, I.; McDonald, G.: A data structure and an algorithm for the nearest point problem. IEEE Transactions on Software Engineering, pp. 631–634, 1983.

[La11] Lange, Dustin; Vogel, Tobias; Draisbach, Uwe; Naumann, Felix: Projektseminar 'Similarity Search Algorithms'. Datenbank-Spektrum, 11(1):51–57, 2011.

[MOV94] Micó, M.L.; Oncina, J.; Vidal, E.: A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. Pattern Recognition Letters, 15(1):9–17, 1994.

[NBZ11] Novak, David; Batko, Michal; Zezula, Pavel: Metric index: An efficient and scalable solution for precise and approximate similarity search. Information Systems, 36(4):721–733, 2011.

[No93] Noltemeier, H. and Verbarg, K. and Zirkelbach, C.: A data structure for representing and efficient querying large scenes of geometric objects: MB* trees. In: Geometric modelling. Springer-Verlag, pp. 211–226, 1993.

[NVZ92] Noltemeier, H.; Verbarg, K.; Zirkelbach, C.: Monotonous Bisector* Trees – a tool for efficient partitioning of complex scenes of geometric objects. Data Structures and Efficient Algorithms, pp. 186–203, 1992.

[NZ14] Novak, David; Zezula, Pavel: Rank aggregation of candidate sets for efficient similarity search. In: Proc. Int. Conf. on Database and Expert Systems Applications (DEXA). Springer, pp. 42–58, 2014.

[Pa99] Patella, Marco: Similarity search in multimedia databases. Dipartmento di Elettronica Informatica e Sistemistica, Bologna, 1999.

[Rh10] Rheinländer, Astrid; Knobloch, Martin; Hochmuth, Nicky; Leser, Ulf: Prefix tree indexing for similarity search and similarity joins on genomic data. In: Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM). Springer, pp. 519–536, 2010.

[Sh77] Shapiro, Marvin: The choice of reference points in best-match file searching. Communications of the ACM, 20(5):339–343, 1977.

[Sk03] Skopal, T.; Pokornỳ, J.; Krátkỳ, M.; Snášel, V.: Revisiting M-tree building principles. In: Advances in Databases and Information Systems. Springer, pp. 148–162, 2003.

[Sk04] Skopal, T.: Pivoting M-tree: A metric access method for efficient similarity search. In: Proc. Dateso Annual Int. Workshop on Databases, Texts, Specifications and Objects (Desna, Czech Republic, April 14-16). pp. 27–37, 2004.

[SMZ15] Sedmidubsky, Jan; Mic, Vladimir; Zezula, Pavel: Face Image Retrieval Revisited. In: Proc. Int. Conf. on Similarity Search and Applications. Springer, pp. 204–216, 2015.

[Tr00] Traina, C.; Traina, A.; Seeger, B.; Faloutsos, C.: Slim-trees: High performance metric trees minimizing overlap between nodes. Advances in Database Technology, EDBT 2000, pp. 51–65, 2000.

[Tr02] Traina Jr, C.; Traina, A.; Faloutsos, C.; Seeger, B.: Fast indexing and visualization of metric data sets using slim-trees. Knowledge and Data Engineering, IEEE Transactions on, 14(2):244–260, 2002.

[Uh91] Uhlmann, J.K.: Satisfying general proximity/similarity queries with metric trees. Information processing letters, 40(4):175–179, 1991.

[Vi86] Vidal, E.: An algorithm for finding nearest neighbours in (approximately) constant average time. Pattern Recognition Letters, 4(3):145–157, 1986.

[Yi93] Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms. Society for Industrial and Applied Mathematics, pp. 311–321, 1993.

[Yi99] Yianilos, P.N.: Excluded middle vantage point forests for nearest neighbor search. In: Proc. 6th DIMACS Implementation Challenge: Near Neighbor Searches (ALENEX). Baltimore, Maryland, USA, 1999.

[Ze06] Zezula, Pavel; Amato, Giuseppe; Dohnal, Vlastislav; Batko, Michal: Similarity Search: The Metric Space Approach. Springer, Berlin, 2006.

[Zh05] Zhou, X.; Wang, G.; Zhou, X.; Yu, G.: BM+-tree: A hyperplane-based index method for high-dimensional metric spaces. In: Database Systems for Advanced Applications. Springer, pp. 398–409, 2005.

[ZS15] Zierenberg, Marcel; Schmitt, Ingo: Optimizing the Distance Computation Order of Multi-Feature Similarity Search Indexing. In: Proc. Int. Conf. on Similarity Search and Applications. Springer, pp. 90–96, 2015.

Dissertation Award Winner

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 507

Query Processing and Optimization in Modern Database Systems

Viktor Leis[1]

Abstract: Relational database management systems, which were designed decades ago, are still the dominant data processing platform. Since then, large DRAM capacities and servers with many cores have fundamentally changed the hardware landscape. As a consequence, traditional database systems cannot exploit modern hardware effectively anymore. This paper summarizes author's thesis, which focuses on the challenges posed by modern hardware for transaction processing, query processing, and query optimization. In particular, we present a concurrent transaction processing system based on hardware transactional memory and show how to synchronize data structures efficiently. We further design a parallel query engine for many-core CPUs that supports the important relational operators including join, aggregation, window functions, etc. Finally, we dissect the query optimization process in the main memory setting and show the contribution of each query optimizer component to the overall query performance.

1 The Architecture of Relational Database Systems

Relational database management systems have stood the test of time and are still the dominant data processing platform. The basic design of these systems stems from the 1980s and was largely unchanged for decades. The core ideas include row-wise storage as well as B-trees on fixed-sized pages backed by a buffer pool, ARIES-style logging, and Two Phase Locking. Recent years, however, have seen many of the design decisions become obsolete due to fundamental changes in the hardware landscape. Before describing the contributions of the author's thesis [Le16a], we give a brief outline of modern database systems and discuss some of the challenges posed by modern hardware.

1.1 Column Stores

After decades of only minor, incremental changes to the basic database architecture, a radically new design, column stores, started to gain traction in the years after 2005. C-Store [St05] (commercialized as Vertica) and MonetDB/X100 [BZN05] (commercialized as Vectorwise) are two influential systems that gained significant mind share during that time frame. The idea of organizing relations by column is, of course, much older [BMK09]. Sybase IQ [MF04] and MonetDB [BQK96] are two pioneering column stores that originated in the 1990s.

Column stores are read-optimized and often used as data warehouses, i.e., non-operational databases that ingest changes periodically (e.g., every night). In comparison with row stores,

[1] Technische Universität München, leis@in.tum.de

column stores have the obvious advantage that scans only need to read those attributes accessed by a particular query resulting in less I/O operations. A second advantage is that the query engine of a column store can be implemented in a much more CPU-efficient way: Column stores can amortize the interpretation overhead of the iterator model by processing batches of rows ("vector-at-a-time"), instead of working only on individual rows ("tuple-at-a-time").

The major database vendors have reacted to the changing landscape by combining multiple storage and query engines in their products. In Microsoft SQL Server, for example, users now can choose between the

- traditional general-purpose row store,
- a column store [La11b] for OnLine Analytical Processing (OLAP), and
- in-memory storage optimized for Online transaction processing (OLTP) [Di13].

Each of these options comes with its own query processing model and specific performance characteristics, which must be carefully considered by the database administrator.

The impact of column stores can be seen in the official benchmark numbers of TPC-H, which is a widely used OLAP benchmark. Before 2011, multiple vendors competed for the TPC-H crown, with the lead changing from time to time between Oracle, Microsoft, IBM, and Sybase. This changed with the arrival of Actian Vectorwise in 2011, which disrupted the incremental "rat race" between the traditional vendors by doubling the reported performance. The dominance of Vectorwise as official TPC-H leader lasted until 2014, when Microsoft submitted new results with their column store engine Apollo [La11b], which has been the leading system in 2016.

1.2 Main-Memory Database Systems

The lower CPU overhead of column store query engines was of only minor importance as long as data was mainly stored on disk (or even SSD). In 2000 one had to pay over $1000 for 1 GB of DRAM[2]. At these prices, any non-trivial database workload resulted in a significant number of disk I/O operations, and main-memory DBMSs—which were a research topic as early as the 1980s [GS92]—were still niche products. In 2008, with the same $1000 one could already buy 100 GB of RAM[3]. This rapid decrease in DRAM prices had consequences for the architecture of database management systems.

Harizopoulos et al.'s paper from 2008 [Ha08] showed that on the—suddenly very common—memory-resident OLTP workloads virtually all time was wasted on overhead like

- buffer management,

[2] DRAM prices are taken from `http://www.jcmit.com/memoryprice.htm`.
[3] The cost continues to decline. At the time of writing, in 2016, the cost was around $4 per GB.

- locking,
- latching,
- heavy-weight logging, and
- an inefficient implementation.

The goal of any database system's designer thus gradually shifted from minimizing the number of disk I/O operations to reducing CPU overhead and cache misses. This lead to a resurgence of research into main-memory database systems. The main idea behind main-memory DBMSs is to assume that all data fits into RAM and to optimize for CPU and cache efficiency. Using careful engineering and by making the right architectural decisions that take modern hardware into account, database systems can achieve orders of magnitude higher performance. Well-known main-memory database systems include H-Store/VoltDB [Ka08, SW13], SAP HANA [Fä11], Microsoft Hekaton [La13], Oracle TimesTen [LNF13], Calvin [Th12], Silo [Tu13], MemSQL, and HyPer [KN11].

1.3 HyPer

The author's thesis [Le16a] has been done in the context of the HyPer project, which started in 2010 [KN10]. HyPer follows some of the design decisions of other main-memory systems (e.g., no buffer manager, no locks, no latches, and (originally) command logging). To avoid fine-grained latches, HyPer also initially followed H-Store's approach of relying on user-controlled, physical partitioning of the database to enable multi-threading.

HyPer has, however, a number of features that distinguish it from many other main-memory systems: From the very beginning, HyPer supported both OLTP and OLAP in the same database in order to make the physical separation between the transactional and data warehouse databases obsolete. Initially, HyPer used OS-supported snapshots [KN11], which were later replaced with a software-controlled Multi-Version Concurrency Control (MVCC) approach [NMK15]. The second unique feature of HyPer is that, via the LLVM [LA04] compiler infrastructure, it compiles SQL queries and stored procedures to machine code [Ne11, NL14]. Compilation avoids the interpretation overhead inherent in the iterator model and thereby enables extremely high performance. LLVM is a widely used open source compiler backend that can generate efficient machine code for many different target platforms, which makes this approach portable. In contrast to previous compilation approaches (e.g., [KVC10]), HyPer compiles multiple relational operators from the same query pipeline into a single intertwined code fragment, which allows it to keep values in CPU registers for as long as possible.

In terms of architecture, most column stores have converged to a similar design [Ab13], which was pioneered by systems like Vectorwise [BZN05] and Vertica [St05]. In-memory OLTP systems, in contrast, show more architectural variety. Compilation is, however, becoming a common building block for OLTP systems, as can be observed by the use of compilation by HyPer [Ne11], Hekaton [Di13], and MemSQL. Other high-performance

systems like Silo [Tu13] also implicitly assume (but do not yet implement) compilation, as the stored procedures are hand-written in C or C++ in these systems. In other areas like concurrency control (e.g., [Tu13] vs. [La11a] vs. [NMK15]), indexing (e.g., [LKN13] vs. [MKM12] vs. [LLS13]), and logging (e.g., [Ma14] vs. physiological) there is much more variety between the systems.

2 The Challenges of Modern Hardware

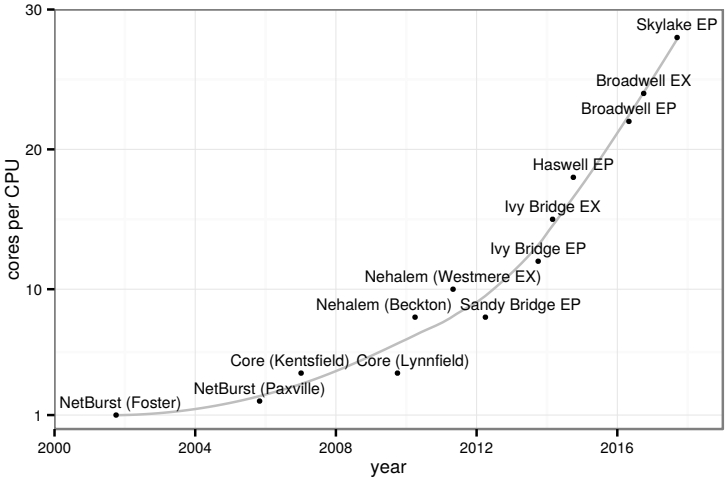


Fig. 1: Number of cores in Intel server processors (largest configuration in each microarchitecture)

Besides increasing main-memory sizes, a second important trend in the hardware landscape is the ever increasing number of cores. Figure 1 shows the number of cores for server CPUs[4]. Over the entire time frame, the clock rate stayed between 2 GHz and 3 GHz and, as a result, single-threaded performance increased only very slowly (by single-digit percentages per year). Note that the graph only shows "real" cores for a single socket. Many servers have 2, 4, or even 8 sockets in a single system and each Intel core nowadays has 2-way HyperThreading. As a result, the affordable and commonly used 2-socket configurations will soon routinely have over 100 hardware threads in a single system. Memory bandwidth has largely kept up with the increasing number of cores and will reach over 100 GB/s per socket with Skylake EP. However, it is important to note that a single core can only utilize a small fraction of the available bandwidth, making effective parallelization essential.

Long before the many-core revolution, high-end database servers often combined a handful of processors—connected by a shared memory bus—in a Symmetric Multi-Processing (SMP) system. Furthermore, database systems have, for a long time, been capable of executing queries concurrently by using appropriate locking and latching techniques. So one

[4] The data is from `https://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors`. For Broadwell EX and Skylake EP server CPUs we show estimates from the press as they were not yet released at the time of writing.

might reasonably ask if any fundamental changes to the database architecture are required at all. Modern hardware, however, has unique challenges not encountered in the past:

Latches are expensive and prevent scaling. Traditional database systems use latches extensively to access shared data structures from concurrent threads. As long as disk I/O operations were frequent, the overhead of short-term latching was negligible. On modern hardware, however, even short-term, uncontested latches can be expensive and prevent scalability. The reason is that each latch acquisition causes cache line invalidations for *all other cores*. As we show experimentally, this effect often prevents scalability on multi-core CPUs.

Intra-query parallelism is not optional any more. For a long time, many systems relied on parallelism from the "outside", i.e., inter-query parallelism. With dozens or hundreds of cores, intra-query parallelism is not an optional optimization because many workloads simply do not have enough parallel query sessions. Without intra-query parallelism, the computational resources of modern servers lie dormant. The widely used PostgreSQL system, for example, will finally introduce (limited) intra-query parallelism in the upcoming version 9.6—20 years after the project started.

Query engines should be designed with multi-core parallelism in mind. Some commercial systems added support for intra-query parallelism a decade ago. This was often done by introducing "exchange" operators [Gr90] that encapsulate parallelism without redesigning the actual operators. This pragmatic approach was sufficient at a time when the degree of parallelism in database servers was low (e.g., 10 threads). To get good scalability on systems with dozens of cores, the query processing algorithms should be redesigned from scratch with parallelism in mind.

Database systems should take Non-Uniform Memory Architecture (NUMA) into account. In contrast to earlier SMP systems, where all processors shared a common memory bus, current systems are generally based on the Non-Uniform Memory Architecture (NUMA). In this architecture each processor has its own memory, but can transparently and cache-coherently access remote memory through an interconnect. Because remote memory accesses are more expensive than local accesses, NUMA-aware data placement can improve performance considerably. Thus, database systems must optimize for NUMA to obtain optimal performance.

Together, these changes explain why traditional systems (e.g., as described in [HSH07]) cannot fully exploit the resources provided today's commodity servers. To utilize modern hardware well, fundamental changes to core database components including storage, concurrency control, low-level synchronization, query processing, logging, etc. are necessary. Database systems specifically designed for modern hardware can be orders of magnitude faster than their predecessors.

3 Contributions

The author's thesis [Le16a] addresses the challenges enumerated above. The solutions were developed within a general-purpose, relational database system (HyPer) and most experiments measure end-to-end performance. Our contributions span the transaction processing, query processing, and query optimization components.

We design a low-overhead, **concurrent transaction processing** engine based on Hardware Transactional Memory (HTM) [LKN14, LKN16]. Until recently, transactional memory—although a promising technique—suffered from the absence of an efficient hardware implementation. Since Intel introduced the Haswell microarchitecture hardware transactional memory is available in mainstream CPUs. HTM allows for efficient concurrent, atomic operations, which is also highly desirable in the context of databases. On the other hand, HTM has several limitations that, in general, prevent a one-to-one mapping of database transactions to HTM transactions. We devise several building blocks that can be used to exploit HTM in main-memory databases. We show that HTM allows one to achieve nearly lock-free processing of database transactions by carefully controlling the data layout and the access patterns. The HTM component is used for detecting the (infrequent) conflicts, which allows for an optimistic—and thus very low-overhead execution—of concurrent transactions. We evaluate our approach on a 4-core desktop and a 28-core server system and find that HTM indeed provides a scalable, powerful, and easy to use synchronization primitive.

While Hardware Transactional Memory is easy to use and can offer good performance, it is not yet widespread. Therefore, we study alternative **low-overhead synchronization** mechanisms for in-memory data structures [Le16b]. The traditional approach, fine-grained locking, does not scale on modern hardware. Lock-free data structures, in contrast, scale very well but are extremely difficult to implement and often require additional indirections. We argue for a middle ground, i.e., synchronization protocols that use locking, but only sparingly. We synchronize the Adaptive Radix Tree (ART) [LKN13] using two such protocols, Optimistic Lock Coupling and Read-Optimized Write EXclusion (ROWEX). Both perform and scale very well while being much easier to implement than lock-free techniques.

We describe the **parallel and NUMA-aware query engine** of HyPer, which scales up to dozens of cores [Le14]. Our "morsel-driven" query execution framework, where scheduling becomes a fine-grained run-time task that is NUMA-aware. Morsel-driven query processing takes small fragments of input data ("morsels") and schedules these to worker threads that run entire operator pipelines until the next pipeline-breaking operator. The degree of parallelism is not baked into the plan but can elastically change during query execution. The dispatcher can react to the execution speed of different morsels but also adjust resources dynamically in response to newly arriving queries in the workload. Furthermore, the dispatcher is aware of data locality of the NUMA-local morsels and operator state, such that the great majority of executions takes place on NUMA-local memory. Our evaluation on the TPC-H and SSB benchmarks shows extremely high absolute performance and an average speedup of over 30 with 32 cores.

We complete the description of HyPer's query engine by proposing a design for the **SQL:2003 window function operator** [Le15b]. Window functions, also known as analytic OLAP functions, have been neglected in the research literature—despite being part of the SQL standard for more than a decade and being a widely-used feature. Window functions can elegantly express many useful queries about time series, ranking, percentiles, moving averages, and cumulative sums. Formulating such queries in plain SQL-92 is usually both cumbersome and inefficient. Our algorithm is optimized for high-performance main-memory database systems and has excellent performance on modern multi-core CPUs. We show how to fully parallelize all phases of the operator in order to effectively scale for arbitrary input distributions.

The only thing more important for achieving low query response times than a fast and scalable query engine is **query optimization**. We shift our focus from the query engine to the query optimizer [Le15a]. Query optimization has been studied for decades, but most experiments were in the context of disk-based systems or were focused on individual query optimization components rather than end-to-end performance. We introduce the Join Order Benchmark (JOB) and experimentally revisit the main components in the classic query optimizer architecture using a complex, real-world data set and realistic multi-join queries. We investigate the quality of industrial-strength cardinality estimators and find that all estimators routinely produce large errors. We further show that while estimates are essential for finding a good join order, query performance is unsatisfactory if the query engine relies too heavily on these estimates. Using another set of experiments that measure the impact of the cost model, we find that it has much less influence on query performance than the cardinality estimates. Finally, we investigate plan enumeration techniques comparing exhaustive dynamic programming with heuristic algorithms and find that exhaustive enumeration improves performance despite the sub-optimal cardinality estimates.

4 Future Work

We have shown that a modern database system that is carefully optimized for modern hardware can achieve orders of magnitude higher performance than a traditional design. However, there are still many unsolved problems, some of which we plan to address in the future.

One important research frontier nowadays lies in supporting mixed workloads in a single database. Many systems that start out as pure OLTP systems, over time add OLAP features thus blurring the distinction between the two system types. While there may be sound technical reasons for running OLTP and OLAP in separate systems, in reality, OLTP and OLAP are more platonic ideals than truly separate applications. One major consequence is that, even for main-memory database systems, the convenient assumption that *all* data fits into RAM generally does not hold. Despite a number of recent proposals, we believe that the general problem of efficiently maintaining a global replacement strategy over relational as well as index data is still not fully solved.

Major changes are also happening on the hardware side and database systems must keep evolving to benefit from these changes. One aspect is the ever increasing number of cores per CPU. While it is not clear whether servers with 1000 cores will be common in the near future—if this indeed happens—it will have a major effect on database architecture. It is a general rule, that the higher the degree of parallelism, the more difficult scalability becomes. Any efficient system that scales up to, for example, 100 cores, will likely require some major changes to scale up to 1000 cores. Thus, some of the architectural decisions may need to be revised if the many-core trend continues.

A potentially even greater challenge is the increasing heterogeneity of modern hardware, which has the potential of disrupting the architecture of database systems. From the point of view of a software developer, in the past, software became faster "automatically" due to increasing clock frequencies. The hardware technologies mentioned above have, however, one thing in common: Programmers have to invest effort to get any benefit from them. Programming for SIMD, GPUs, or FPGAs is very different (and more difficult) than using the instruction set of a conventional, general-purpose CPU. A database system will have to decide which part of a query should be executed on which device, all while managing the data movement between the devices and the energy/heat budget of the overall system. Put simply, no one currently knows how to do this and no doubt it will take considerable research effort to find a satisfactory solution.

Whereas the technologies mentioned above promise faster computation, new storage technologies like PCIe-attached NAND flash and non-volatile memory (NVM) like Phase Change Memory threaten to disrupt the way data is stored and accessed. In order to avoid changing the software stack much, it is certainly possible to hide modern storage devices behind a conventional block device interface. However, this approach leaves performance on the table as it ignores the specific physical properties like the block erase requirement of NAND flash or the byte-addressability of non-volatile memory. Thus, research is required to find out how to best utilize these new storage technologies.

Finally, even the venerable field of query optimization still has many unsolved problems. One promising approach is to rely more heavily on sampling, which is much cheaper than in the past when CPU cycles were costly and random disk I/O would have been required. Sampling, for example across indexes, opens up new ways to estimate the cardinality of multi-way joins [Le17], which after decades of research, is still done naively in most systems.

References

[Ab13] Abadi, Daniel; Boncz, Peter A.; Harizopoulos, Stavros; Idreos, Stratos; Madden, Samuel: The Design and Implementation of Modern Column-Oriented Database Systems. Foundations and Trends in Databases, 5(3), 2013.

[BMK09] Boncz, Peter A.; Manegold, Stefan; Kersten, Martin L.: Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. PVLDB, 2(2), 2009.

[BQK96] Boncz, Peter A.; Quak, Wilko; Kersten, Martin L.: Monet And Its Geographic Extensions: A Novel Approach to High Performance GIS Processing. In: EDBT. 1996.

[BZN05] Boncz, Peter; Zukowski, Marcin; Nes, Niels: MonetDB/X100: Hyper-Pipelining Query Execution. In: CIDR. 2005.

[Di13] Diaconu, Cristian; Freedman, Craig; Ismert, Erik; Larson, Per-Åke; Mittal, Pravin; Stonecipher, Ryan; Verma, Nitin; Zwilling, Mike: Hekaton: SQL server's memory-optimized OLTP engine. In: SIGMOD. 2013.

[Fä11] Färber, Franz; Cha, Sang Kyun; Primsch, Jürgen; Bornhövd, Christof; Sigg, Stefan; Lehner, Wolfgang: SAP HANA database: data management for modern business applications. SIGMOD Record, 40(4), 2011.

[Gr90] Graefe, Goetz: Encapsulation of Parallelism in the Volcano Query Processing System. In: SIGMOD. 1990.

[GS92] Garcia-Molina, Hector; Salem, Kenneth: Main Memory Database Systems: An Overview. IEEE Trans. Knowl. Data Eng., 4(6), 1992.

[Ha08] Harizopoulos, Stavros; Abadi, Daniel J.; Madden, Samuel; Stonebraker, Michael: OLTP through the looking glass, and what we found there. In: SIGMOD. 2008.

[HSH07] Hellerstein, Joseph M.; Stonebraker, Michael; Hamilton, James R.: Architecture of a Database System. Foundations and Trends in Databases, 1(2), 2007.

[Ka08] Kallman, Robert; Kimura, Hideaki; Natkins, Jonathan; Pavlo, Andrew; Rasin, Alex; Zdonik, Stanley B.; Jones, Evan P. C.; Madden, Samuel; Stonebraker, Michael; Zhang, Yang; Hugg, John; Abadi, Daniel J.: H-store: a high-performance, distributed main memory transaction processing system. PVLDB, 1(2), 2008.

[KN10] Kemper, Alfons; Neumann, Thomas: HyPer - Hybrid OLTP&OLAP High Performance Database System. Technical report, TUM, 2010.

[KN11] Kemper, Alfons; Neumann, Thomas: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE. 2011.

[KVC10] Krikellas, Konstantinos; Viglas, Stratis; Cintra, Marcelo: Generating code for holistic query evaluation. In: ICDE. 2010.

[LA04] Lattner, Chris; Adve, Vikram: LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In: CGO. Mar 2004.

[La11a] Larson, Per-Åke; Blanas, Spyros; Diaconu, Cristian; Freedman, Craig; Patel, Jignesh M.; Zwilling, Mike: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. PVLDB, 5(4), 2011.

[La11b] Larson, Per-Åke; Clinciu, Cipri; Hanson, Eric N.; Oks, Artem; Price, Susan L.; Rangarajan, Srikumar; Surna, Aleksandras; Zhou, Qingqing: SQL Server column store indexes. In: SIGMOD. 2011.

[La13] Larson, Per-Åke; Zwilling, Mike; ; Farlee, Kevin: The Hekaton Memory-Optimized OLTP Engine. IEEE Data Eng. Bull., 36(2), 2013.

[Le14] Leis, Viktor; Boncz, Peter; Kemper, Alfons; Neumann, Thomas: Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In: SIGMOD. 2014.

[Le15a] Leis, Viktor; Gubichev, Andrey; Mirchev, Atanas; Boncz, Peter; Kemper, Alfons; Neumann, Thomas: How Good Are Query Optimizers, Really? PVLDB, 9(3), 2015.

[Le15b] Leis, Viktor; Kundhikanjana, Kan; Kemper, Alfons; Neumann, Thomas: Efficient Processing of Window Functions in Analytical SQL Queries. PVLDB, 8(10), 2015.

[Le16a] Leis, Viktor: Query Processing and Optimization in Modern Database Systems. Disseration, Technische Universität München, 2016.

[Le16b] Leis, Viktor; Scheibner, Florian; Kemper, Alfons; Neumann, Thomas: The ART of practical synchronization. In: DaMoN. 2016.

[Le17] Leis, Viktor: Cardinality Estimation Done Right: Index-Based Join Sampling. In: CIDR. 2017.

[LKN13] Leis, Viktor; Kemper, Alfons; Neumann, Thomas: The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In: ICDE. 2013.

[LKN14] Leis, Viktor; Kemper, Alfons; Neumann, Thomas: Exploiting hardware transactional memory in main-memory databases. In: ICDE. 2014.

[LKN16] Leis, Viktor; Kemper, Alfons; Neumann, Thomas: Scaling HTM-Supported Database Transactions to Many Cores. IEEE Trans. Knowl. Data Eng., 28(2), 2016.

[LLS13] Levandoski, Justin J.; Lomet, David B.; Sengupta, Sudipta: The Bw-Tree: A B-tree for new hardware platforms. In: ICDE. 2013.

[LNF13] Lahiri, Tirthankar; Neimat, Marie-Anne; Folkman, Steve: Oracle TimesTen: An In-Memory Database for Enterprise Applications. IEEE Data Eng. Bull., 36(2), 2013.

[Ma14] Malviya, Nirmesh; Weisberg, Ariel; Madden, Samuel; Stonebraker, Michael: Rethinking main memory OLTP recovery. In: ICDE. 2014.

[MF04] MacNicol, Roger; French, Blaine: Sybase IQ Multiplex - Designed For Analytics. In: VLDB. 2004.

[MKM12] Mao, Yandong; Kohler, Eddie; Morris, Robert Tappan: Cache craftiness for fast multicore key-value storage. In: EuroSys. 2012.

[Ne11] Neumann, Thomas: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB, 4(9), 2011.

[NL14] Neumann, Thomas; Leis, Viktor: Compiling Database Queries into Machine Code. IEEE Data Eng. Bull., 37(1), 2014.

[NMK15] Neumann, Thomas; Mühlbauer, Tobias; Kemper, Alfons: Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In: SIGMOD. 2015.

[St05] Stonebraker, Michael; Abadi, Daniel J.; Batkin, Adam; Chen, Xuedong; Cherniack, Mitch; Ferreira, Miguel; Lau, Edmond; Lin, Amerson; Madden, Samuel; O'Neil, Elizabeth J.; O'Neil, Patrick E.; Rasin, Alex; Tran, Nga; Zdonik, Stanley B.: C-Store: A Column-oriented DBMS. In: VLDB. 2005.

[SW13] Stonebraker, Michael; Weisberg, Ariel: The VoltDB Main Memory DBMS. IEEE Data Eng. Bull., 36(2), 2013.

[Th12] Thomson, Alexander; Diamond, Thaddeus; Weng, Shu-Chun; Ren, Kun; Shao, Philip; Abadi, Daniel J.: Calvin: fast distributed transactions for partitioned database systems. In: SIGMOD. 2012.

[Tu13] Tu, Stephen; Zheng, Wenting; Kohler, Eddie; Liskov, Barbara; Madden, Samuel: Speedy transactions in multicore in-memory databases. In: SOSP. 2013.

Industrial Program

Big Data

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 521

SAP HANA Vora: A Distributed Computing Platform for Enterprise Data Lakes

Christian Sengstock[1], Christian Mathis[2]

Abstract

Businesses are increasingly leveraging the power of Big Data to improve their services and products. We call the infrastructure to process and manage the heterogenous kinds of data their "data lakes". Data lakes are used to store and process massive streams of sensor data, service data, collected or generated media, archived enterprise data, and massive transactional databases, among others. Such infrastructures are often realized by Hadoop clusters and low-cost persistence layers, such as S3 or SWIFT data stores.

SAP HANA Vora is a distributed computing platform that sits on top of Data Lakes and was developed to build a basis layer for upcoming Big Data applications in the enterprise. It provides high-performance in-memory data processing and management capabilities, is easily extensible by new computing engines, extends the existing Big Data software stack, and integrates with the existing enterprise IT by design. We present an architectural overview of the system.

[1] SAP SE, 69190 Walldorf, c.sengstock@sap.com
[2] SAP SE, 69190 Walldorf, christian.mathis@sap.com

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 523

Big Data is no longer equivalent to Hadoop in the industry

Andreas Tönne[1]

Abstract: For a long time, industry projects solved big data problems with Hadoop. The massive scalability of MapReduce algorithms and the HBase database brought solutions to an unanticipated level of computing. But this obstructs the view for the need of change. Business goals that emerge from Industry 4.0 or IoT have long been addressed with a suboptimal architecture. New business goals require a rethinking of the big data architecture instead of being driven by the known Hadoop ecosphere. We discuss the transformation of a Hadoop-centric middleware solution to a streaming architecture from a business value perspective. The new architecture also replaces a single NoSQL database by polyglot persistence that allows to focus on best performance and quality of each data processing step. We also discuss alternative architecture approaches like Lambda that were evaluated in the course of the transformation. We show that a single technology choice likely leads to a solution that is suboptimal.

1 Extended Abstract

Depending on whether you are a researcher or practitioner, Hadoop was created in 2002 as Yahoo's research solution to a better (scaling) search engine, called project Nutch. Or it really came to life when it was promoted top-level project of the Apache Software Foundation in 2008. Also in 2008 Cloudera was founded to offer a commercial version of Hadoop. This was one critical first step towards maturity that made Hadoop usable for CIOs around the world. Hadoop offered an unanticipated level of computing power to corporates and a promise of endless, massive scalability both for storage in HDFS/HBase and computing using the map-reduce concept of Google (2004). Many companies started out with Hadoop, investigating its usefulness for business problems from a purely technological perspective. But the concepts of Hadoop were too "alien" for the traditionally conservative corporate world to immediately relate them to business value. And although the term "big data" has been around for more than two decades, the big data hype peaked around 2012/2013. At this time, big data = Hadoop was firmly rooted in the heads of corporate IT.

Many companies (57%) that have invested in the Hadoop technology suffered from pains growing the needed skills or recruiting the needed experts, and a similar percentage of companies (49%) is still trying to figure out how to get value out of their Hadoop investment (Gartner Hadoop Adoption Study 2015), and this matches very well with our experience. This situation leaves corporate IT stuck with a Hadoop installation and the related promises, desperately trying to monetize the investment. Since Hadoop is simply a massively scalable technology to process big data amounts in an asynchronous way in a distributed data lake, new hypes like data science (analytics), complex event processing or deep learning, while not matching the processing paradigm of Hadoop, are nonetheless sought to yield

[1] dibuco GmbH, Franz-Schubert Str. 75, 70195 Stuttgart, andreas.toenne@dibuco.de

the promised business value. But these new promises expose a mismatch between the architecture underlying the Hadoop technology and many business goals that are to be solved with analytics and machine learning concepts. These new business goals may emerge not only from hypes like Industry 4.0 or IoT, or also as a response strategy to digital disruptors, which are threatening many businesses. Common to these business values is the aspect of time (i.e., batch versus stream processing). Often it is the case that the business value of big data deteriorates over time, requiring near realtime analysis or data pattern recognition. For example, machine data, online payment processing or business statistics of an online platform cannot wait for a long running Hadoop batch to complete. These goals require a rethinking of the big data strategy and the adoption of more modern big data solutions. The current technology trends are streaming platforms like Apache Storm or Apache Spark Streaming and inmemory processing. Also the Lambda architecture by Nathan Marz that combines batch style processing (data lake) and event processing (data stream) into a common view is growing in popularity. Yet in business meetings we still observe the identification of big data storage with HBase as the only choice. We also see resistance of IT to admit further big data technologies to their technology stack because "we already have Hadoop in operation".

In order to break this assumed equality of big data and Hadoop, IT needs to stop wagging the dog by the tail. It is unfortunately common to drive a big data project from the technology end. This is wrong and ultimately harms the customers! We need to start with the business goals and work our way down to technology and will discover that we have an exponentially growing number of big data technologies to choose from. In our talk we demonstrate this way of evolving a Hadoop centric big data solution to a more capable and even more scalable streaming solution on a customer product development project that we were involved with in the last three years. We highlight the importance of polyglot persistence to choose both the right data model and the right database technology to achieve the business goals. Very important was also the question whether more data is actually better from a business value perspective. Finally, if one tries to address every problem related to concurrency and consistency with Hadoop batches, one may discover that one simply runs out of time to schedule the batches. The lesson learned is that a premature single big data storage and processing technology choice might lead to a solution that is suboptimal for many business goals.

Data Lake

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 527

Drying up the data swamp – Vernetzung von Daten mittels iQser GIN Server

Florian Pfleiderer[1]

Abstract

In vielen Unternehmen laufen heute heterogene Daten aus vielfältigen Quellen in Data Lakes zusammen, die immer mehr zu Data Swamps verkommen. Oft ist nicht bekannt, was sich in den zahlreichen Datentöpfen befindet und in welcher Qualität die Daten tatsächlich vorliegen. Typische Big Data Technologien wie zum Beispiel Hadoop alleine bieten kaum eine Möglichkeit, diesem Chaos Herr zu werden. Immer mehr Firmen zeigen daher Interesse an kompletten Lösungen, statt eigene Lösungen aufwändig aus einzelnen Technologien zusammen zu stellen. Die iQser GmbH entwickelt mit dem GIN Server eine solche Lösung, die unterschiedliche Ansätze des Data Engineering kombiniert, um verschiedenste Problemstellungen der semantischen Datenanalyse lösen zu können.

Um aus einem Mix von strukturierten und unstrukturierten Daten Informationen gewinnen zu können, werden Daten und Dokumente basierend auf ihren Inhalten mithilfe qualifizierter Relationen automatisch vernetzt. Der hierbei entstehende Graph ist die Basis für die Schöpfung von neuem Wissen aus vorher unbekannten Daten. Solche Daten können nicht immer im Vorfeld klassifiziert oder auf bestimmte Arten modelliert werden, da hierfür das notwendige a-priori Wissen über die Inhalte der Daten fehlt oder zu aufwändig zu erlangen ist. Dies betrifft insbesondere die Erstellung von Ontologien im Sinne des Semantic Web oder Open Linked Data. Hier geht die Lösung von iQser einen anderen Weg und erzeugt in einem Bottom-Up-Ansatz aus den Daten selbst ein Modell über eine automatische semantische Vernetzung.

In dem Vortrag wird erklärt, welche Ziele mit der Entwicklung des GIN Servers verfolgt wurden, um Ordnung in einem Data Swamp zu schaffen, in dem mehr Daten nicht immer mehr Nutzen bedeuten, weil es immer schwerer wird diese zu korrelieren und ordnen zu können. Es wird darauf eingegangen, welchen Herausforderungen man sich bei der Entwicklung einer solchen Lösung stellen muss, welche Erfahrungen gemacht und Erkenntnisse hierbei gewonnen wurden und warum ein Schritt weg von einer Batch-Verarbeitung und hin zu einem Streaming-basierten Ansatz es der Anwendungsarchitektur ermöglicht hat, Ziele besser zu erreichen.

[1] dibuco GmbH, Franz-Schubert Str. 75, 70195 Stuttgart, florian.pfleiderer@dibuco.de

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 529

Möglichkeiten und Grenzen von Textanalytics im eCommerce

Laura Hoyden[1] Frank Rosenthal[2] Jonas Hausruckinger[3]

Abstract

Die Synchronisation von Produktstammdaten ist eine zentrale Aufgabe im eCommerce, denn schließlich muss sichergestellt werden, dass wirklich die Variante des Artikels ausgeliefert wird, die ein Kunde bestellt hat. Schlüsselsysteme wie EAN/GTIN und UPC leisten dabei einen guten Beitrag, lösen das Problem in der Praxis aber bei weitem nicht vollständig, da viele Handelsgüter erst gar nicht mit herstellerunabhängigen Schlüsseln gekennzeichnet sind oder z.B. Packungsgrößen und Varianten zu häufig verwechselt werden. Die Abbildung von Produktstammdaten anhand der Artikelbezeichnungen ist daher eine Aufgabe für moderne Textanalyseverfahren an der Grenze von strukturierten und natürlichsprachlichen Ansätzen. Als Marktforschungsunternehmen setzt GfK solche Textanalyseverfahren ein, um unterschiedliche Artikelbezeichnungen für eine Produktvariante zu erkennen und für Marktberichte zu normieren.

Im ersten Teil des Vortrags wird erklärt, wie ein proprietäres Machine Learning Verfahren genutzt werden kann, um den Suchraum für die Zuordnung einer unbekannten Artikelvariante so einzuschränken, dass ein Bearbeiter mit einem Blick die passendste von nur wenigen wahrscheinlichen Varianten wählen und bestätigen kann. Zentral dafür sind ein mehrstufiger Prozess sowie komplementäre Verfahren (Ensemble) in einzelnen Schritten, da Textquellen von unterschiedlicher Güte verarbeitet werden müssen, aber auch die Größe des Suchraums von Warengruppe zu Warengruppe stark differiert.

Im zweiten Teil der Präsentation werden die Voraussetzungen und Herausforderungen untersucht, die für eine vollautomatische Zuordnung zu schaffen bzw. zu lösen sind. Im Kern steht dabei die Frage, wie man die qualifizierten Vorschläge mit einem intervall-normierten Konfidenzmaß versehen kann. Über einen Schwellwert soll dann eine automatische Zuordnung möglich sein, wobei entscheidend ist, dass eine sehr hohe Zuordnungsgüte erreicht wird.

Der Vortrag arbeitet mit vielen konkreten Beispielen aus der Warenwelt von Gebrauchsgütern und gibt einen Einblick in die Praxis eines datengetriebenen Unternehmens, welches Business Intelligence als Produkt anbietet.

Zielgruppe der Veranstaltung sind an Textanalytics interessierte Zuhörer, die idealerweise entsprechende Grundkenntnisse mitbringen.

[1] GfK SE, Nordwestring 101, 90419 Nürnberg, Laura.Hoyden@gfk.com
[2] GfK SE, Nordwestring 101, 90419 Nürnberg, Frank.Rosenthal@gfk.com
[3] GfK SE, Nordwestring 101, 90419 Nürnberg, Jonas.Hausruckinger@gfk.com

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 531

Autonomous Data Ingestion Tuning in Data Warehouse Accelerators

Knut Stolze[1] Felix Beier[1] Jens Müller[1]

Abstract: The IBM DB2 Analytics Accelerator (IDAA) is a state-of-the art hybrid database system that seamlessly extends the strong transactional capabilities of DB2 for z/OS with very fast processing of OLAP and analytical SQL workload in Netezza. IDAA copies the data from DB2 for z/OS into its Netezza backend, and customers can tailor data maintenance according to their needs. This copy process, the *data load*, can be done on a whole table or just a physical table partition. IDAA also offers an *incremental update* feature, which employs replication technologies for low-latency data synchronization.

The accelerator targets big relational databases with several TBs of data. Therefore, the data load is performance-critical, not only for the data transfer itself, but the system has to be able to scale up to a large number of tables, i. e., tens of thousands to be loaded at the same time, as well. The administrative overhead for such a number of tables has to be minimized.

In this paper, we present our work on a prototype, which is geared towards efficiently loading data for many tables, where each table may store only a comparably small amount of data. A new load scheduler has been introduced for handling all concurrent load requests for disjoint sets of tables. That is not only required for a multi-tenant setup, but also a significant improvement for attaching an accelerator to a single DB2 for z/OS system. In this paper, we present architecture and implementation aspects of the new and improved load mechanism and results of some initial performance evaluations.

1 Introduction

The IBM® DB2® Analytics Accelerator for z/OS (IDAA, cf. Fig. 1) [Ba15] is an extension for IBM's® DB2® for z/OS® database system. Its primary objective is the extremely fast execution of complex, analytical queries on a snapshot of the data copied from DB2. Many customer installations have proven that the combination of DB2 with a seamlessly integrated IDAA delivers an environment where both, transactional workload and analytical queries, is supported without impacting existing and new applications. The achieved query acceleration for analytical workloads is at least an order of magnitude, often even exceeding that.

After the initial version of IDAA based on a Netezza backend [Fr11] became available in 2011, functional enhancements were added to expand the product's scope. The first steps were more fine-granular data synchronization mechanisms like *partition load* and *incremental update*. A completely new use case was introduced with the *high performance storage saver*, which transformed the accelerator into a cost-efficient archiving solution [St13]. More recently, IDAA was enhanced to directly support analytical workload. Such workload

[1] IBM Germany Research & Development GmbH, Schönaicher Straße 220, 71032 Böblingen, Germany, {stolze,febe,jens.mueller}@de.ibm.com

Fig. 1: System Overview of the IBM DB2 Analytics Accelerator (IDAA)

often involves complex data transformation, which can now be processed completely within the accelerator by means of accelerator-only tables [BSM16]. DML operations like `INSERT`, `UPDATE`, and `DELETE` statements can be send from a customer's application to the DB2 for z/OS server, which sends the complete SQL statement on to the accelerator for execution. No exchange of the data between DB2 for z/OS and the IDAA server is necessary and, thus, frees resources on System z. Furthermore, analytics stored procedures of the IBM® Netezza® Analytics (INZA) product are enabled directly on the accelerator [IB14b] to add support for data mining workloads as well.

Nevertheless, data movement and copying the data from DB2 for z/OS to the accelerator is still, and will remain, one of the most important aspects of the appliance. Applications are working with tables in DB2 for z/OS, and changes made there need to be made available on the accelerator. IDAA's customers have many tables on the system – 30,000 tables or more is not a rare situation. Many of these tables are rather small in size, i. e., empty or just a few rows up to a few MBs worth of data.

In this paper, we present our work to improve IDAA in order to optimize batch-loading of large table sets. We developed a prototype, which considers all currently pending load requests of a table batch and autonomously determines an optimal schedule for the next time slice. The degree of parallelism is adjusted, based on the current system utilization. That takes into account the currently running loads as well as any other workload on the Netezza system, such as analytic queries or maintenance operations.

The remainder of the paper is structured as follows. The architecture of the IBM DB2 Analytics Accelerator is revisited in Sect. 2. Sect. 3 explains the new architecture for IDAA's data load. It covers the new load scheduler and its interactions with other components in the system. Some initial performance results are presented in Sect. 4, and the paper concludes with a summary in Sect. 5.

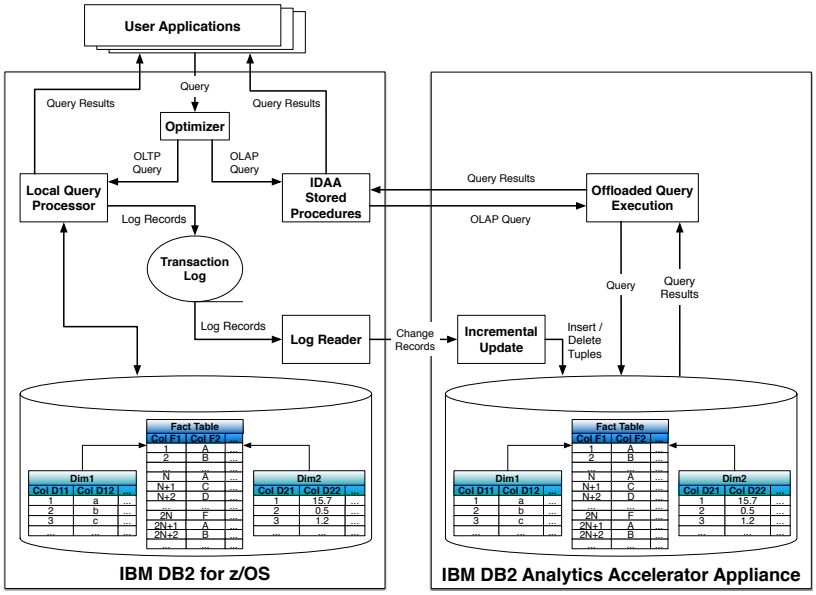


Fig. 2: IDAA System Architecture and its Integration into DB2 for zOS

2 Overview on the IBM DB2 Analytics Accelerator

The IBM DB2 Analytics Accelerator (IDAA) [Ba15] is a hybrid system, which uses DB2® for z/OS® [IB14a] for transactional workload with the well-known excellent performance characteristics. A copy of the DB2 data resides in the accelerator, which is based on Netezza technology [Fr11], and deals with analytical workload in an extremely high-performing way. The DB2 optimizer is responsible for the query routing decision, i. e., whether to run the query in DB2 itself or to offload it to the accelerator.

The benefits of this system are a reduced complexity of the necessary IT infrastructure on customer sites for executing both types of workloads and its tight integration into the existing DB2 for z/OS system, which results in overall cost reductions for customers. A single system is used and not a whole zoo of heterogenous platforms needs to be operated and maintained. Aside from the system management aspects, investments into a business' applications is protected, which is crucial for companies of a certain size. Existing applications can continue to use DB2 unchanged, while additionally exploiting the analytics capabilities of IDAA without any (or only minuscule) changes.

2.1 High-level System Architecture

IDAA provides the data storage and SQL processing capabilities for large amounts of data with exceptional query performance. Fig. 2 illustrates the high-level architecture. The key is the seamless integration of all components in DB2 to leverage the Netezza appliance as analytical backend. SQL statements are passed from DB2 to IDAA, which drives the execution in Netezza. Administrative requests, e. g., to provide a list of accelerated tables, to trigger table load operations, or to setup automatic log-based data replication for offloaded

tables, are handled in IDAA itself in conjunction with the IDAA stored procedures. Either way, the external interface to work with IDAA is DB2. If necessary, SQL queries against Netezza are executed to collect backend-related meta data and/or statistical information which are utilized for the load scheduler presented in Sect. 3.1.

Fig. 3 shows that it is possible to associate multiple accelerators with a single DB2 system in order to establish an environment that supports high availability and disaster recovery. Appropriate workload balancing is applied by DB2 in case the connected Netezza backends have different hardware and workload characteristics. Similarly, an accelerator implements a multi-tenant environment where different DB2 systems can connect to it, sharing its resources. Another workload balancing layer is applied by the Netezza backend on the SQL level to cover multi-tenancy scenarios.

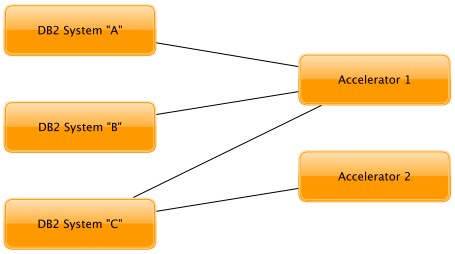


Fig. 3: System Setup with M:N Connections between DB2 z/OS Subsystems and IDAA Appliances

Such a setup is very flexible and allows our customers to slowly grow the exploitation of IDAA, depending on the (changing) workload. However, our experience has shown that the accelerator can easily become overloaded, or the DB2 systems cause contention to such a degree that SLAs can no longer be satisfied. While Netezza's workload management, combined with IDAA's spill-to-disk functionality for buffering query results in slow receiver scenarios, is excellent for concurrent query processing, the data load processing requires more resources and leaves room for improvements which will be presented in the following.

2.2 Data Replication Strategies

IDAA offers three options for refreshing the data that has been shadow-copied from DB2. Entire tables or individual table partitions can be refreshed in the accelerator batch-wise. Fig. 4 and 5 illustrate both bulk-loading scenarios conceptually. The DB2 table is on the left side, and the IDAA table on the right. The sketch above the arrow shows which pieces of the DB2 table are copied to the IDAA table.

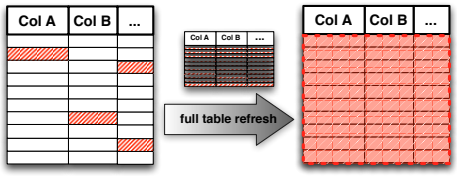


Fig. 4: Full Table Refresh

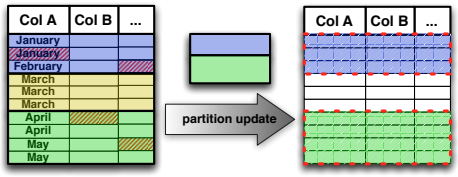


Fig. 5: Partition Update

For tables having a higher update frequency and where a high data currency is desired, the Incremental Update feature (cf. Fig. 6) is suitable. This feature uses IBM® InfoSphere® Change Data Capture (CDC) [IBM13, Be12], which reads DB2 transaction logs and extracts all changes to accelerated DB2 tables. Unlike the bulk-load strategies, CDC replicates only those change data records while unchanged data is not copied.

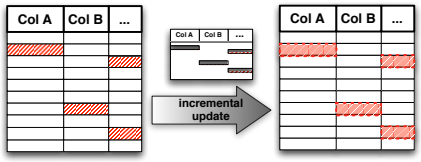


Fig. 6: Refreshing Table Data with Incremental Update

While the bulk-load strategies offer great throughputs but require to copy large data volumes and, hence, are high-latency operations, the incremental update strategy replicates only low data volumes with larger overheads per tuple, but a low latency until change records have been applied in the backend. A latency of about 1 minute is achieved, which is usually fast enough for reporting systems and data warehouses, considering that complex accelerated queries may take several minutes (or hours) in DB2 vs. a few seconds only with IDAA. Furthermore, getting really *the* last committed changes for business reports is not that crucial as long as the report is not based on hours old data. It is the responsibility of the database user/administrator to trigger the refresh of the data in each accelerator individually (or to set up incremental update where appropriate) [IB13]. Since many customers have thousands of tables in DB2 and also in IDAA, it is important to understand query access patterns to the individual tables and analyzing the accelerated workload with the help of monitoring tools.

2.3 Load Processing

Right from the very first version of IDAA, the development team focussed on a highly efficient load process for the data synchronization between DB2 for z/OS and IDAA with its Netezza backend. Traditional replication technologies like Q-Rep or CDC [IBM13] were evaluated, which were still separate products in year 2011. We found that replication products were not even close to delivering the desired throughput of $1TB/h$. Therefore, IDAA implemented its own batch-load mechanism, which is depicted in Fig. 7.

On z/OS side, DB2's `UNLOAD` utility is employed to extract the data verbatim from the pages of the database system. It bypasses most of the relational data processing components of DB2. One of the performance improvements for IDAA was to expose the DB2-internal record layout, which avoids conversions to some external format. The UNLOAD utility writes the data to a Unix Systems Services (USS) named pipe so that no persistent storage is needed. The IDAA stored procedure `ACCEL_LOAD_TABLES`, running in DB2 for z/OS, reads the data from the named pipe and sends it directly on to the IDAA. A light-weight variation of the DRDA protocol [DRD03] is used for the communication, in particular to exchange control information.

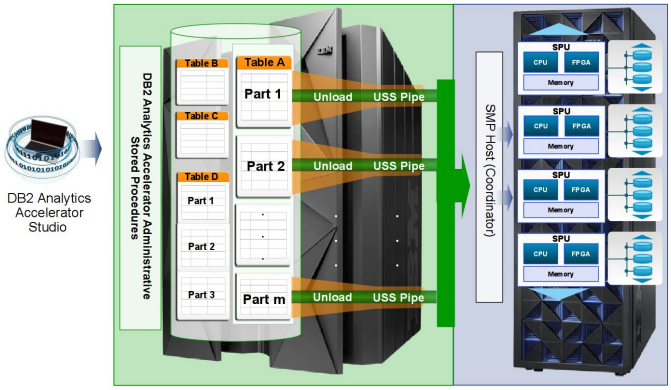


Fig. 7: Traditional Load Architecture

On IDAA server side, the coordinator receives the data and writes it into a named pipe. Again, copying of the data is minimized and no persistent storage is used. The Netezza backend reads from the named pipe when inserting the data into its table. Netezza does the only data conversion from the DB2-internal row format[4] to its own format and to write it to data pages.

Such an end-to-end pipeline is called *load stream*. If a table is partitioned in DB2, multiple partitions are processed in parallel by instantiating multiple load streams. Non-partitioned tables use a single load-stream only. The customer can configure the degree of parallelism, which is the maximum number of concurrent load streams. That configuration applies to a single invocation of the `ACCEL_LOAD_TABLES` stored procedure.

This design delivered a throughput of $1TB/h$ right from the start. The aforementioned support for the DB2-internal row format, and avoiding copying and data transformation steps within DB2 for z/OS and IDAA gave another boost in later versions. Now the bottleneck is the network bandwidth between DB2 and IDAA. Combining (bonding) active and failover network lines resulted in an end-to-end throughput of more than $4TB/h$ under lab conditions.

3 Autonomous Load Performance Tuning

Despite the excellent performance numbers, the load architecture described in Sect. 2.3 does have some caveats. Tables holding a large volume of data are required to fully saturate a load stream. When the data volume is too small, the overhead for establishing a load stream becomes more and more noticeable.

In one situation, a customer wanted to synchronize several thousand, mostly empty or nearly empty tables using a single `ACCEL_LOAD_TABLES` stored procedure call. The total elapsed time exceeded the available time in the nightly batch window. Sequentially loading the tables led to a mostly under-utilized system. The solution was straight-forward: determine

[4] Netezza was extended for this IDAA scenario to understand the DB2-internal row format.

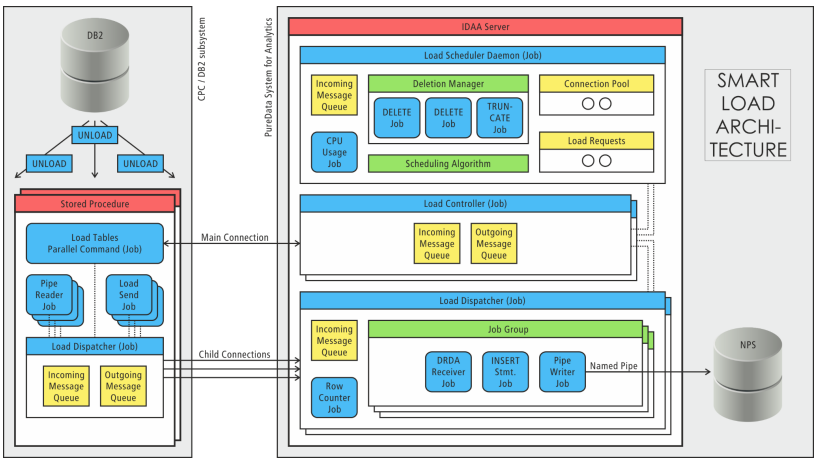


Fig. 8: Smart Load Architecture

clusters of tables to load together, and use multiple concurrent stored procedure calls. Still, such an approach cannot be called *easy* to use. And the customer would still have to take care of managing the parallelism for the concurrent stored procedure calls and, in addition, across different DB2 systems that are connected to the same accelerator.

3.1 Smart Load Architecture

The new design, called *Smart Load* is shown in Fig. 8. The central change (compared to Fig. 7) is the introduction of the *Load Scheduler Daemon* on the IDAA server. It is responsible for managing and scheduling all concurrently running load operations.

The *Load Controller* handles a single `ACCEL_LOAD_TABLES` stored procedure call and the tables specified therein. The controller determines all its work units. A work unit is an unpartitioned table or a partition of a partitioned table. A work unit descriptor contains (amongst other details) information about the data volume. All work unit descriptors are passed to the load scheduler, who adds them to its pool of load requests.

The scheduler itself implements an optimization algorithm to determine the order in which the remaining work units shall be processed. Different state-of-the-art algorithms can be used for that.We have implemented a simple bin packing approach. The key criteria for the specific algorithm is that it produces an initial result very fast and subsequently improves on that even with a partially changed set of work units.

In addition to defining the sequence in which work units shall be processed, the load scheduler monitors the current system utilization of the accelerator. The following three classic cases are handled:

Under-utilization The scheduler determines that there are still sufficient resources available to start a new load stream. Parallelism increases due to that, and load throughput increases as well.

Over-utilization The accelerator is under pressure in terms of available resources. In order to remedy the situation, the load scheduler reduces the parallelism. That means, when the next work unit finishes, the load stream used for it will not start processing a new work unit. Instead, it is kept as a passive load stream[5] until parallelism can be increased again or some other load controller relinquishes one or more of its load streams at its end.

Normal utilization The system is neither over-utilized nor does it have sufficient free resources to start another load stream (or no work unit requiring another load stream is pending). The parallelism setting remains unchanged.

The system utilization is determined by monitoring CPU, I/O, and network utilization. The resources typically used for a single load stream are determined from currently or previously running load operations. Additionally, the current throughput of active load requests is considered as another factor for reducing the number of active load streams. Hence, the algorithm to determine whether a new load stream can be started, is simple and robust. A condition – like the following for CPU utilization – is applied for all relevant measures.

$$current\_CPU\_Utilization + expected\_Load\_Stream\_CPU\_Utilization < 100\%$$

The decision to increase the parallelism and starting another load stream is made jointly with the *Load Dispatcher* running on the DB2 for z/OS side. The load dispatcher monitors the system utilization there. So if the load scheduler wants to allow a controller to start a new load stream, the controller informs the dispatcher, and only if the dispatcher agrees will the load stream be started. In case the dispatcher disagrees, the load scheduler may pick another controller to use the available resources. Thus, we have implemented a distributed scheduling and workload balancing algorithm specifically tailored to IDAA's needs. Since we measure general performance metrics, our approach also takes concurrently running (non-load) workload into account – without introducing any dependencies on the code level.

Note that it is possible that oscillation effects may appear, i. e., if the scheduler increases parallelism due to free resources, the next calculation may detect an under-utilization, and so on. That is not a problem here, however. First, the load scheduler only makes a decision whether to increase the parallelism by 1 (one), which prevents large spikes. So even if an over-utilization occurs, it only exceeds the available resources by a small amount. Second, changes in the degree of parallelism only become active when new work units are added to the scheduler's pool or when a work unit has finished its data transfer. Therefore, the changes themselves are relatively rare. Third, once scheduled, transfer operations are never aborted due to a change in parallelism which avoids unnecessary restart operations.

This approach not only increases the efficiency for batched table loads but also significantly reduces administration complexity. Customers merely have to state once which tables

[5] In the past, a load stream was created for each partition of table. That includes the creation of the TCP/IP connection (denoted as child connection in Fig. 7) as well as creating and starting the various threads. Creating the TCP/IP connection and threads once and pooling them for reuse gave a significant improvement on its own already for the small tables scenario.

shall be loaded and synchronized without the need to manage multiple stored procedure invocations to exploit parallelism. Additionally, our system automatically tunes itself to the best resource exploitation. There is no need to set some configuration parameter or to do some initial calibration. The load scheduler continuously calibrates itself automatically.

4 Preliminary Performance Evaluation

We used our prototype to run some initial tests to gain a better feel for the improvements. The test used an older (and rather slow) Netezza system as backend.

First, we loaded 50 unpartitioned tables, and each table stored only a single row. A single row is the minimum possible amount of data per table because the IDAA load process has an optimization for empty tables where it skips all data transfer and processing. In particular, no DB2 `UNLOAD` utility is started.

Tab. 1 shows the measured execution times. Even if the times may vary depending on the actual data, the results clearly show that tearing down and re-establishing the load stream for each work unit (as was done in the past) adds a significant overhead for such small-sized tables. Furthermore, loading all tables sequentially looses a lot of potential, showing that inter-table parallelism is important. Extrapolating the improvement of about 20x from our 50 tables to 20,000 tables results in several hours of savings.

Scenario	Load Time
Smart Load	13s
Old (sequential) Load	4min 16s

Tab. 1: 50 Tables with 1 row each

Tables with just a single row (or very few rows) are extreme cases – not uncommon for our customers, but not the norm either. That's why, we also tested a TPC-H scenario with a scale factor of 30 MB. The improvements are not as dramatic as above because data transfer and processing take a larger slice. Still, the results in Tab. 2 show an improvement by factor 6.

Scenario	Load Time
Smart Load	7s
Old (sequential) Load	42s

Tab. 2: TPC-H Workload

5 Summary

In this paper we have presented a prototype that extends the IBM DB2 Analytics Accelerator with smart load scheduling capabilities. It simplifies the usage of the accelerator for data maintenance via batch loading dramatically. Customers no longer have to manage the resources consumed on the accelerator across different connected DB2 systems. Instead,

IDAA actively monitors all systems involved and automatically scales the parallel degree for processing load requests. The net result is an overall improvement for throughput of batch load operations. We expect a lot of satisfaction from the much simplified user experience.

Next steps for our work will be the integration of the prototype into the product. More intensive performance measurements will be forthcoming. We expect that different scheduling algorithms may show some impact, but given the typical pattern of data load, it is unlikely to observe a major impact. Nevertheless, we intend to verify our assumption empirically.

6 Trademarks

References

[Ba15] Baumbach, Ute; Becker, Patric; Denneler, Uwe; Hechler, Eberhard; Hengstler, Wolfgang; Knoll, Steffen; Neumann, Frank; Schoellmann, Guenter Georg; Souissi, Khadija; Zimmermann, Timm: . Accelerating Data Transformation with IBM DB2 Analytics Accelerator for z/OS. IBM Redbooks, 2015.

[Be12] Beaton, A.; Noor, A.; Parkes, J.; Shubin, B.; Ballard, C.; Ketchie, M.; Ketelaars, F.; Rangarao, D.; Tichelen, W.V.: . Smarter Business: Dynamic Information with IBM InfoSphere Data Replication CDC. IBM Redbooks, 2012.

[BSM16] Beier, Felix; Stolze, Knut; Martin, Daniel: Extending Database Accelerators for Data Transformations and Predictive Analytics. In: Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016. S. 706–707, 2016.

[DRD03] The Open Group. DRDA V5 Vol. 1: Distributed Relational Database Architecture, 2003.

[Fr11] Francisco, P.: . The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. IBM Redbooks, 2011.

[IB13] IBM: Sychronizing data in IBM DB2 Analytics Accelerator for z/OS. Bericht, IBM, 2013. http://www-01.ibm.com/support/docview.wss?uid=swg27038501.

[IB14a] IBM: . DB2 11 for z/OS, 2014. http://www.ibm.com/support/knowledgecenter/api/content/SSEPEK/db2z_prodhome.html.

[IB14b] IBM: . IBM Netezza Analytics – In-Database Analytics Developer's Guide, Release 3.0.1, 2014.

[IBM13] IBM. IBM InfoSphere Data Replication V 10.2.1 documentation, 2013.

[St13] Stolze, Knut; Köth, Oliver; Beier, Felix; Caballero, Carlos; Li, Ruiping: Seamless Integration of Archiving Functionality in OLTP/OLAP Database Systems Using Accelerator Technologies. Datenbanksysteme fÃ¼r Business, Technologie und Web (BTW) 2013 : Tagung vom 11. - 15. MÃ¤rz 2013 in Magdeburg, S. 383–402, March 2013.

Trends

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 543

Database Management Systems: Trends and Directions

Namik Hrle[1]

Abstract

Business Analytics, Big Data, Systems of Engagement, IoT and Cloud delivery model create new requirements that profoundly affect database management systems technology. Columnar orientation, in-memory databases, no-SQL stores, Spark and Hadoop integration are the trends that have already proven their values in some of the most challenging application workloads. Hybrid systems promise converging of transactional and analytical processing and enable new way of driving insight from data, fueling new or significantly enhanced business models. At the same time, traditional, relational database management systems are still in foundations of a large majority of mission critical, core business applications. Where do traditional DBMS offerings fit within these new technology trends and business requirements? What is database providers' strategy to remain relevant under the new conditions? These questions will be addressed in this keynote presentation which will discuss bringing together all data in all paradigms (transactional, analytical, unstructured, etc.) with the goal of "making data simple" to consume.

[1] IBM Deutschland GmbH, IBM-Allee 1, 71139 Ehningen, hrle@de.ibm.com

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 545

SAP HANA – The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads

Norman May[1], Alexander Böhm[2], Wolfgang Lehner[3]

Abstract: The journey of SAP HANA started as an in-memory appliance for complex, analytical applications. The success of the system quickly motivated SAP to broaden the scope from the OLAP workloads the system was initially architected for to also handle transactional workloads, in particular to support its Business Suite flagship product. In this paper, we highlight some of the core design changes to evolve an in-memory column store system towards handling OLTP workloads. We also discuss the challenges of running mixed workloads with low-latency OLTP queries and complex analytical queries in the context of the same database management system and give an outlook on the future database interaction patterns of modern business applications we see emerging currently.

1 Introduction

The increase of main memory capacity in combination with the availability of multi-core systems was the technical foundation for the In-Memory SAP HANA database system. Based on the SAP Business Warehouse Accelerator (SAP BWA) product, SAP HANA combined a column-based storage engine with a row-store based query engine. As a result, it delivered superb runtime numbers for analytical workloads by fully leveraging the performance of the columnar data organization as well as algorithms highly tuned for in-memory processing (cache awareness, full SIMD support etc.).

The success of the disruptive approach taken by SAP HANA, to move towards a highly parallel and in-memory computing model for data management also triggered to re-think and finally shake the foundations of traditional enterprise data management architectures: While the traditional separation of transactional and analytical processing has clear advantages for managing and specializing the individual systems, it also comes with constraints which cannot be tolerated to serve modern business applications: For example, data on the analytical side is only updated periodically, often only on a nightly basis resulting in stale data for the analytical side. Complex ETL processes have to be defined, tested, and maintained. Data preparation tasks from executing ETL jobs, populating data warehouse databases as well as deriving Data Marts to adhere to specialized physical storage representations of BI tools etc. reflect a highly error-prone as well as time-consuming task within the enterprise data management stack. Last but not least, two separate database management systems need to be licensed and operated.

[1] SAP SE, 69190 Walldorf, norman.may@sap.com
[2] SAP SE, 69190 Walldorf, alexander.boehm@sap.com
[3] TU Dresden, Fakultät Informatik, 01062 Dresden, wolfgang.lehner@tu-dresden.de

Inspired by the potential of SAP HANA, Hasso Plattner started to formulate the vision of an "Enterprise Operational Analytics Data Management System" combining OLTP and OLAP workloads as well as the underlying data set based on a columnar representation ([Pla09]). While this approach was lively discussed within academia as well as openly questioned by senior database researchers ([SC05]) declaring mantra-like that "One Size does not Fit All!", SAP took on the challenge to investigate how far the envelope could be pushed. The goal was to operate one single system providing the basis for realtime analytics as part of operational business processes while also significantly reducing TCO. Specifically, key questions from a technical perspective have been:

- What are the main capabilities of the SAP HANA columnar storage engine, and how could they be exploited for transactional workloads?
- What characteristics of transactional workloads have to be treated with specific optimizations within the SAP HANA engine, and what would be technical solutions?
- How could transactional as well as analytical workload co-exist from a scheduling as well as data layout perspective?

Within this paper, we highlight some technical challenges and give key solutions with respect to these questions. We will show, how the SAP HANA system evolved and continues to evolve from a purely analytical workhorse to an integrated data management platform that is able to serve the largest ERP installations with 5 million queries/h and more than 50K users concurrently. Figure 1 outlines the major evolutionary steps we present in this paper in a chronological order.

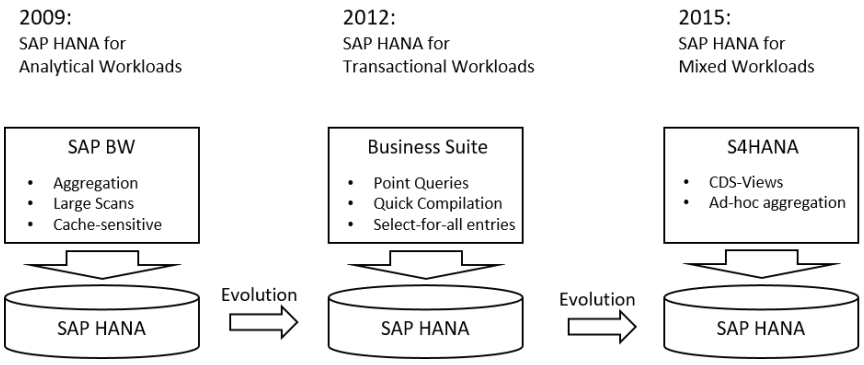


Fig. 1: Evolution of SAP HANA.

Structure of the Paper

The next section summarizes the core ingredients of SAP HANA to provide high performance for analytical workloads (Section 2). The next step within the SAP HANA evolution is then addressed within Section 3 outlining some challenges of supporting transactional query patterns. Section 4 finally shares some requirements and solutions of supporting

S/4HANA, the new SAP flagship product, implementing the vision of a single point of truth for all business related data and any type of interaction with this data set. Section 5 finally concludes with hinting at some challenges as well as opportunities for academia to be tackled within the near future.

2 SAP HANA for Analytical Scenarios

As mentioned above, the original use-case of SAP HANA was to support analytical workloads, motivated by the requirements of the SAP Business Warehouse (BW). The original design therefore heavily focused on read-mostly queries with updates performed in bulks (as the result of ETL processes) and took several architectural choices that strictly favor fast query processing over efficient update handling. Aggregation queries typically addressed star- or snowflake schemas with wide dimension tables and very few large fact tables. Concurrency was limited to a few (in the range of hundreds) users with heavy read-intensive workloads.

2.1 Parallelization at all Levels

One of the core design principles of SAP HANA to cope with these requirements was to perform parallelization at all levels starting from hardware (e.g. heavily using vectorized processing) to query execution. This choice was motivated by the recent trends in hardware development, where CPU clock speed does no longer increase significantly with newer hardware generations, but an ever growing number of CPUs becomes available instead. Consequentially, to make use of an ever growing number of CPUs, SAP HANA parallelizes the execution of queries at different levels:

- Inter-Query Parallelism by maintaining multiple user sessions.
- Intra-Query Parallelism/Inter-Operator Parallelism by concurrently executing operations within a single query.
- Intra-Operator Parallelism using multiple threads for individual operators.

Figure 2 shows the principle of one of the most frequently used operators for analytical workloads–aggregation [TMBS12, MSL+15]: A central (fact) table is logically subdivided into different fragments which may fit into the processor cache. The aggregation algorithm first aggregates chunks of the tuples into thread-local hash tables that fit into the L2 cache. When a certain number of pre-aggregated data is available those tables are merged into the final aggregated results. In this second reduction phase we avoid locking the buckets of hash tables by partitioning the groups to separate threads. Overall, this leads to almost linear scalability of the aggregation operator with the number of available threads. Other relational operators are tuned in a similar fashion for a high-degree of parallelism to exploit modern hardware capabilities.

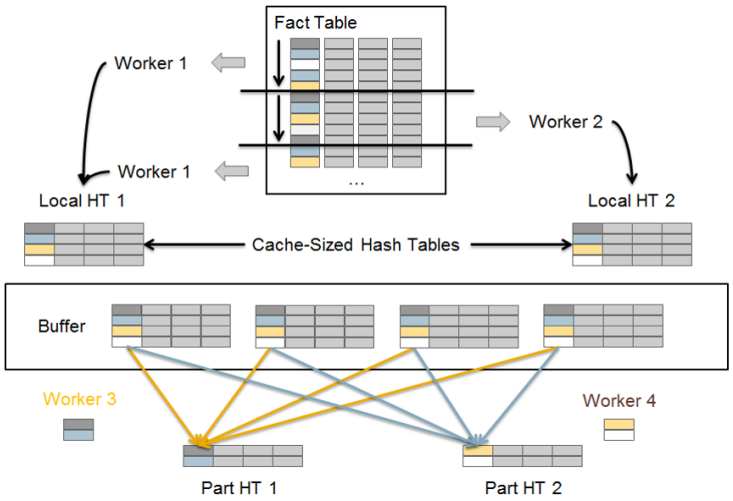


Fig. 2: Principle of parallel aggregation within SAP HANA.

2.2 Scan-friendly Physical Data Layout

SAP HANA relies heavily on a column-oriented data layout, which allows to scan only those columns which are touched by a query. Moreover, since dictionary compression is used for all data types within SAP HANA, the stored column values can be represented by the minimal number of bits required to encode the existing number of distinct values [FML+12]. This storage layout both enables a high compression potential (e.g. run-length encoding, sparse encodings etc.) and allows to fully exploit the capabilities of the CPU's hardware prefetcher to hide the latency induced by the memory access. The dictionary itself is sorted with respect to the actual column values supporting an efficient value lookup for example to resolve search predicates. The columnar storage layout and dictionary compression are also used in other database systems, e.g. [IGN+12, KN11, RAB+13, LCF+13, LCC+15], but using ordered dictionaries for all data within an in-memory database system is a unique feature of SAP HANA.

To avoid re-encoding large amounts of data upon dictionary changes, update operations are buffered within a special data structure (delta index) to keep the main part of the table as static (and highly compacted) as long as possible. An explicit merge operation fuses the buffered entries with the static part of the table and creates a new version of the main part to keep the data in a scan-friendly format.

2.3 Advanced Engine Capabilities

In addition to plain SQL support, SAP HANA supports analytical applications by providing a rich bouquet of specialized "engines". For example, a planning engine provides primitives

to support financial planning tasks like disaggregation of global budgets to different cost centers according to different distribution characteristics. A text engine delivers full text retrieval capabilities from entity resolution via general text search techniques to classification algorithms. Special engines like geospatial, timeseries, graph, etc. in combination with a native connectivity with R as well as machine learning algorithms using the AFL extension feature of SAP HANA complement the built-in services for analytical applications.

In addition to engine support, the SAP HANA database engine is part of the SAP HANA data management platform offering integration points with other systems for example to implement federation via the "Smart Data Access" (SDA) infrastructure or specialized connectivity to Hadoop and Spark systems. "Dynamic Tiering" (DT) is offered to perform data aging transparently for the application. DT allows to keep hot data (actuals) within SAP HANA's main memory and cold data (historical data) within SAP IQ by providing a single system experience (e.g. single transactional context, single user base, single admin console) [MLP+15].

2.4 Summary

SAP HANA's DNA consists in mechanisms to efficiently run analytical workloads for massive data sets stored within SAP HANA's columnar data representation. In order to support analytical workloads beyond traditional single engine capabilities, SAP HANA offers a rich set of add-ons to act as a solid and high performance foundation for data analytics applications.

3 OLTP Optimizations in SAP HANA

The second step within the evolutionary process of SAP HANA was positioned to provide a solid foundation for SAP's OLTP workload. One particular goal was to support SAP's ERP product based on the enterprise-scale ABAP stack which includes the ABAP programming language as well as the associated application server, and application lifecycle management. While academia heavily questioned the possibility to optimize a column-store for OLTP workloads, the goal of SAP was to provide a robust solution to sustain a typical enterprise workload in the range of 50.000 OLTP statements/second, comprising queries as well as concurrent updates. Within this section, we highlight some of the optimizations we could integrate into the productive version of SAP HANA without compromising the analytical performance, and also highlight performance improvements that were made over time.

3.1 Initial Analysis and Resulting Challenges

Since SAP HANA comes with a column as well as a row store, the obvious solution would be to internally replicate transactional data and use the row store for OLTP workloads. While this approach has been followed by other systems (e.g. Oracle 12c [LCC+15], IBM

DB2 [RAB+13] and MS SQL Server [LCF+13]), it would be impractical for ERP scenarios with more than 100.000 tables from a management as well as an overall cost perspective (as this replication-based approach significantly increases the overall memory requirements of the system by storing data twice). Consequently, the column store-based implementation has to be advanced to deal with update intensive scenarios, where insert/update/delete operations are performed on a fine granularity (i.e. primary key access as the most relevant access path compared to scan-based data access), data retrieval typically performed for all attributes in a row (`SELECT *` requires row reconstruction out of individual column values), and typically a very high number of concurrently active users requiring a robust resource management.

Based on those observations, a magnitude of optimizations has been realized with SAP HANA; some of the key optimizations applied to productive SAP HANA are discussed in more detail in the subsequent sections.

3.2 Short-Running Queries

While the original query execution model was designed for long-running, scan-based queries, OLTP is characterized by short-running queries. As query compilation time does not much affect the overall runtime of OLAP-style queries, query preparation may be the dominant factor for short-running queries. In many cases, compiling an OLTP query can simply not be tolerated. Therefore, SAP HANA stores optimized SQL statements in a plan cache which associates the SQL string with a compiled query execution plan. Consequently SQL statements are not even parsed when a corresponding entry exists in the plan cache. For parametrized queries, the SQL statement is optimized again and cached when the first parameter value is passed for execution. Plan cache entries are either invalidated when DDL operations are performed that may invalidate the precompiled plan or after a certain number of executions to accommodate for updates to the data.

To emphasize the importance of plan caching we refer to Figure 3. In this figure, we have sorted the top 100 statements of two large productive instances of SAP HANA running an ERP workload based on accumulated query execution time and plot them by their execution count. In both systems, the ten most frequently executed statements comprise one third of all statement executions in the system. The plan cache in these systems consumes a few GB out of a few TB of main memory to guarantee cache hit ratios above 99%, i.e. less than 1% of all incoming queries require a compilation phase. In the two ERP systems shown in Figure 3, cached query execution plans are executed a few thousand times. However, if compilation is necessary, a query of "simple nature", e.g. single table access based on the primary key, does not even enter the full-blown query optimization path but is parametrized and directly executed using specialized access code.

While maximal parallelization within the database engine is the key for OLAP-style queries, plan-level parallelism is disallowed for OLTP queries in order to avoid context switches, scheduling overhead, and the resulting cache misses. The optimizer only allows parallelism,

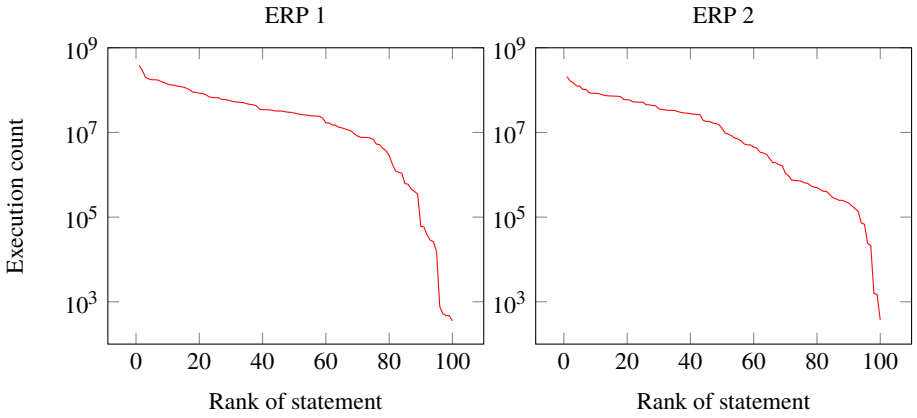


Fig. 3: Execution count of TOP 100 queries in two large productive ERP systems.

if (almost) linear speedup can be expected based on the current load in the system [PSM+15]. A further investigation reveals that parallelization in OLTP scenarios is already implicitly done at the application server level with more than 10.000 concurrent users.

3.3 Optimized Physical Data Representation

OLTP scenarios–especially in the context of SAP ERP–very often exhibit "empty attributes" typically showing only one single default value. The major reason for this is that not all business functionality (i.e. extensions for specific industries) that is backed by corresponding fields in a table is used by all customers, but anyhow fields are retrieved by the default access paths encoded in the application server stack. Consequentially, in productive ERP systems, we often find more than one thousand columns that store only a single (default) value – including several large and hot tables in functional components of the ERP like financials, sales and distribution, or material management. Access on such columns is optimized by completely bypassing the columnar representation and storing the single default values within the "header" of the columnar table, i.e. within the in-memory storage. This reduces the latching overhead when accessing these columns. In internal OLTP benchmarks on ERP data sets we measured about 30% higher throughput and 40% reduced CPU consumption when using this optimization.

As mentioned in the previous section, the columns of a table are stored in delta and main structures to buffer changes to the data and periodically apply these changes by generating a new version of the main part. Those maintenance procedures can be easily weaved into the loading procedures of data warehouse systems. Necessary database reorganizations can thus be directly performed as part of the ETL processes. For OLTP, exclusively locking a table generates massive lock contention because the system continuously has to accept update operations. As a consequence, reorganizations must not interfere with or even completely block concurrently running OLTP queries. Within SAP HANA, the MVCC scheme is extended to explicitly support delta merge operations requiring a table lock only

for switching over to the new version resulting in a very limited impact on concurrent workload.

Finally, queries with a predictable result set cardinality have an optimized physical result vector representation in row-format. This format reduces one allocation per column vector to a single allocation of a compact array. Costly cache misses are thus avoided when materializing the query result because of the dense memory representation.

3.4 Transaction Management

SAP HANA relies on snapshot isolation in two variants: In statement-level snapshot isolation every statement receives a new snapshot and a statement reads all versions that were committed when the statement started. This is the default mode of snapshot isolation used in SAP HANA. With transaction-level snapshot isolation all statements of a transaction read the latest versions committed before the transaction started or that were modified by the transaction itself.

The initial transaction manager of SAP HANA was tracking visibility information via an explicit bit-vector per transaction on table level. Read transactions with the same snapshot shared this bit-vector, but write transactions generated a new bit-vector which was consistent with their snapshot. Bit-vectors representing snapshots of transactions that were older than the snapshot of the oldest open transaction could be removed by a garbage collector. While this design is feasible for OLAP workload with rare updates spanning only a few tables within a single (load) transaction, it turned out to be too expensive and memory consuming for transactions with only a few update operations per table, touching multiple tables sprinkled across the whole database.

This deficit for OLTP workload resulted in a significant extension of the transaction manager and a row-based visibility tracking mechanism. For every row the timestamp when it was created or deleted is stored for the row (note that updates create a new version of the row). As soon as all active transactions see a created row or likewise no active transaction may see a deleted row, these timestamps can be replaced by a bit indicating its visibility. Furthermore, rows that are not visible to any transaction can be removed during the next delta merge operation. This change in the design allows to handle fine-granular updates and is independent of the number of concurrently running transactions. At the same time, OLAP workloads still benefit from the efficient bit-vector-based visibility test for most rows.

Further improvements of the garbage collection are possible as detailed in [LSP+16]. Especially, in mixed workloads, OLTP applications trigger many new versions of data objects while long-running OLAP queries may block garbage collection based on pure timestamp comparison. The SAP HANA garbage collector improves the classical garbage collection based on timestamp-based visibility by 1) identifying versions that belong to the same transaction, 2) identifying tables that are not changed by old snapshots, and 3) identifying intervals of snapshots without active transactions. Combining these garbage collection strategies resulted in reduced memory consumption for keeping versions of

records for multiple snapshots. For the TPC-C benchmark the latency of statements could be reduced by 50%. When having mixed workloads with concurrent long-running statements the OLTP throughput of TPC-C was about three times higher with the improved garbage collection compared to the commonly used global garbage collection scheme.

3.5 Latching and Synchronization

Especially on large systems with many sockets, synchronization primitives that protect internal data structures can become a major scalability bottleneck. These latches require cross-socket memory traffic which is factors slower than the cache-conscious code that is protected by these latches. In many cases, the access to the protected data structure is dominated by read operations, e.g. column access or B-tree access. Consequently, HANA carefully optimizes those accesses balancing the complexity of the implementation with the gain in performance. This confirms the work by [LSKN16].

Latching on the column level is needed to protect against unloading of columns in the presence of memory pressure [SFA+16] or installing a new version of a column after a delta merge operation. This scenario is compatible with RCU-based latching, i.e. update operations install a new version of the object while existing readers either keep working on old references and new readers immediately work on the new version of the column [McK04]. In the case of unloading columns, the column appears as unloaded for new readers while the unloading becomes effective only after the last reader dereferenced the column. In the case of delta merge operations a new version of the merged column is installed in parallel to the old version. The merge operation switches the versions of the main fragment and also of the delta fragments in an atomic operation, and thus with minimal impact on both read and update operations.

As a last optimization HANA uses Intel TSX operations to increase scalability. These operations are especially important in the B-tree used for the lookup of value IDs in the delta fragment because the delta fragment is accessed both for read operations and also updated concurrently by any update operation. In [KDR+14] it is shown that using TSX results in far better scalability with the number of concurrent threads. This is especially important on systems with a growing number of available cores as we see them already today in deployments of SAP HANA.

3.6 Application-Server/DBMS Co-Design

As already presented in [Böh15], SAP HANA exploits the potential to optimize across boundaries within the software stack. Obviously, SAP HANA natively optimizes for the ABAP language and runtime.

On the one hand, the "Fast Data Access" (FDA) feature allows to directly read from and write into a special internal data representation which can be shared between the database system and the application server. Bypassing the SQL connectivity stack for data transfer

results in a 20% speedup on average (Figure 4a) depending obviously on the cardinality of the data set.

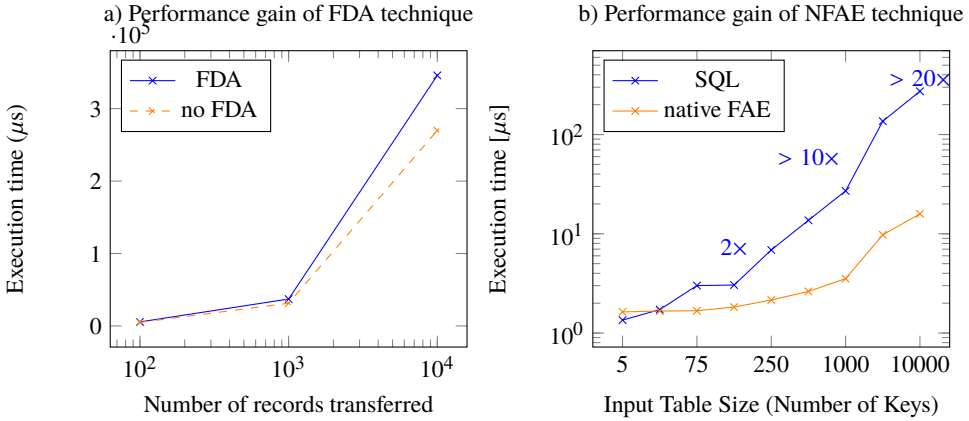


Fig. 4: Impact of SAP HANA optimizations with respect to FDA and NFAE.

On the other hand, the "Native For All Entries" (NFAE)-technique modifies the ABAP runtime to convert a sequence of conjunctions, disjunctions or large IN-lists into a semi-join, which can be more efficiently executed by the underlying database system. For example, the ABAP statement below extracts train departure and arrival information for all trains with an ID and a specific date given within the runtime table `key_tab` residing in the application server.

```
SELECT t.departure, t.arrival FROM trains t
INTO CORRESPONDING FIELDS OF TABLE result_tab
FOR ALL ENTRIES IN key_tab k
WHERE t.ID = k.c1 AND t.DATE = k.c2
```

NFAE exploits FDA to pass the runtime table of the application server efficiently to the SAP HANA process and issues a semi-join between the database table `trains` and the (former) runtime object `key_tab`. Depending on the cardinality of the runtime table, a speedup in the order of a magnitude can be achieved without changing a single line of application code (Figure 4b).

3.7 Results

As outlined, evolving the columnar storage engine and runtime towards an OLTP capable system required substantial changes in all components from transaction management to special support of the SAP application server. To underpin the achievements, Figure 5 quantifies the performance improvements of some of the most critical DB operations (log scale!). The measurements comprise a 2-year period and are derived from the productive SAP HANA code base.

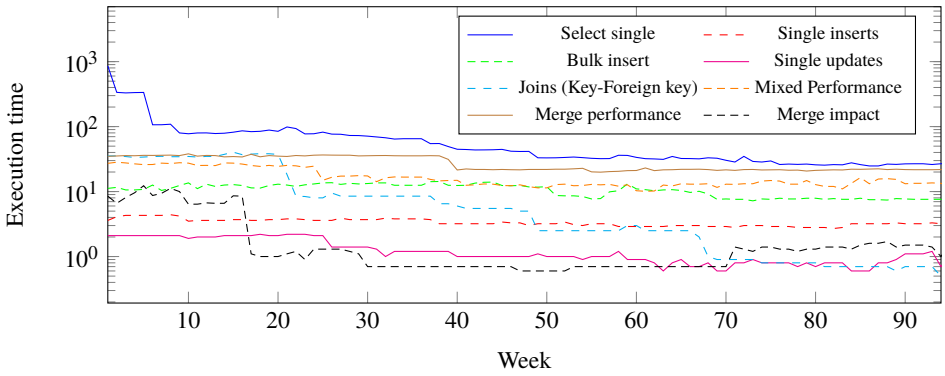


Fig. 5: Impact of SAP HANA optimizations to improve OLTP workload.

As can be seen, point queries (select single) have improved by over an order of magnitude, which also applies for the impact of delta merge operations on concurrently running database workloads. Dramatic improvements have been achieved also for key-foreign key joins, which are–in addition to point queries–the most performance critical query patterns for OLTP workloads.

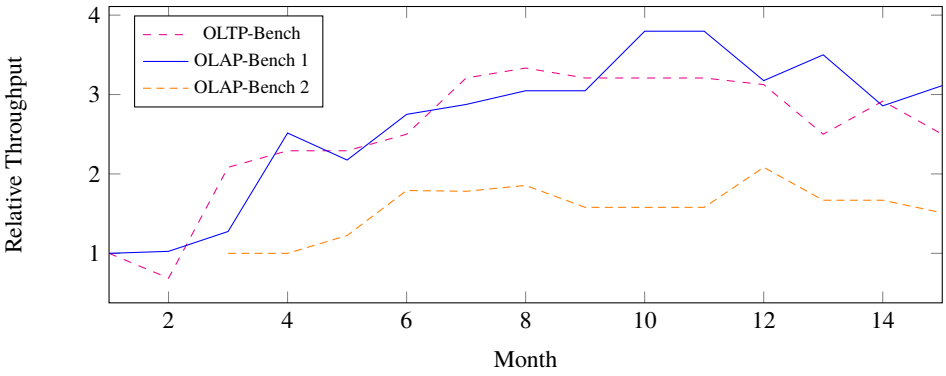


Fig. 6: Performance evolution of application scenarios.

Figure 6 presents how the throughput for two analytical and one transactional application scenarios improved over a 15 month period. Due to legal reasons we present only relative throughput numbers here: For the OLAP benchmarks the throughput is measured in queries per hour, and for the transactional throughput it is measured in the number of concurrent clients which execute transactional queries and updates with a bounded response time. The experiments were executed on a 4TB system with 16 sockets and logical 480 cores. For the analytical benchmarks the number of queries per hour processed by the system increased by up to 4 times. At the same time the number of concurrent users that could be handled by SAP HANA while guaranteeing a maximal response time per query also improved by a factor of three.

In total, the improvements presented so far resulted in

- a factor 25+ efficiency gain in highly concurrent OLTP select operations.
- a factor 15+ improved runtime performance for short-running join operations.
- a factor 8+ improved performance in order-to-cash benchmark.
- no congestion/queuing situations due to delta merge operations.

Overall, SAP HANA provides more than "good-enough" throughput for OLTP operations for typical ERP scenarios, although the system was a priori not designed as a highly specialized OLTP engine but took most design choices in favor of superior analytics performance. Most notably, the above-mentioned optimizations could be achieved without compromising the OLAP performance of the system.

4 SAP HANA Optimizations for Mixed Workloads

As already motivated in the introduction, the success of SAP HANA triggered a re-thinking of business processes and business application support. In order to efficiently operate a modern enterprise, the technically required decoupling of transactional and analytical scenarios can no longer be tolerated. In order to close the gap between making and executing operational decisions based on analytical evaluations and (more and more) simulations, the tight coupling of these two worlds based on a common data management foundation is becoming a must-have. In addition to application requirements, the accompanying operational costs of running two separate systems is no longer accepted.

As a consequence, SAP HANA was extended to deal with mixed workloads on the same database instance. This implies that a specific schema for analytical scenarios (e.g. star-/snowflake schemas) are no longer the base of executing OLAP workloads. Instead, the system must be able to (a) schedule different types of queries (OLTP as well as OLAP) with significantly different resource requirements as well as (b) efficiently run OLAP-style queries on-top of an OLTP-style database schema.

The following sections share some insights into different optimization steps applied to the productive version of SAP HANA.

4.1 Resource Management for Mixed Workloads

A major challenge of mixed workloads we see in SAP HANA are thousands of concurrent transactional statements in addition to hundreds of concurrent analytical queries. The former class of statements requires predicable and low response times, but this is hard to achieve as complex concurrent analytical queries acquire significant amounts of CPU and memory for a real-time reporting experience. Additionally, complex ad-hoc queries may turn out to be

far more resource-intensive than expected, and such statements must not block business critical transactional operations.

One solution to achieve robustness for transactional workload while serving concurrent analytical workload is to assign resource pools to these workloads. However, it is known that this leads to sub-optimal resource utilization [Liu11], and thus is adverse to reducing operational costs. Consequently, instead of over-provisioning ressources, SAP HANA detects cases of massive memory consumption and aborts the associated analytical statements. This is considered acceptable because given a proper sizing of the system such statements should be exceptional.

Likewise, SAP HANA adapts the concurrency of a statement based on the recent CPU utilization [PSM+15]. This means that on a lightly loaded server an analytical query can use all available CPU resources. Under heavy CPU load, i.e. when many statements are processed concurrently, the benefit of parallelization vanishes, and consequently most statements use very few threads avoiding the overhead of context switches. Considering NUMA architectures with 16 sockets or more and about 20 physical cores per socket it also turns out that it is most efficient to limit the concurrency of a statement to the number of cores per socket. This nicely complements the processing of statements on the sockets where the accessed data is allocated [PSM+15].

As a final principle for scheduling mixed workloads with SAP HANA, a statement is classified as OLTP style within the compilation phase (potentially supported by hints coming directly from the application server). For query processing, such transactional statements are prioritized in order to achieve a guaranteed response time, which is crucial for interactive ERP applications. Otherwise, OLAP queries would dominate the system and queuing effects would result in significantly delaying (short running) OLTP queries [WPM+15].

Figure 7 quantifies the results of our optimizations for NUMA-aware scheduling of mixed workloads executed on the same system as the experiment of figure 6. In this figure we plot how the throughput of a typical OLTP scenario on the x-axis versus the throughput of a complex analytic workload on the y-axis improved over time relative to the base line in SAP HANA Rev. 70.

4.2 Heavy on-the-fly aggregation

Due to limited database performance, the ERP application code maintained a huge number of redundant aggregate tables reflecting business-logic specific intermediate results like "sales by region" or even pre-aggregates ("total sales per year") [Pla14]. This application architecture allowed on the one hand to lookup aggregate values within OLTP operations but showed some significant drawbacks on the other hand:

- The application gets more complex because it needs to maintain aggregates besides implementing actual business logic.

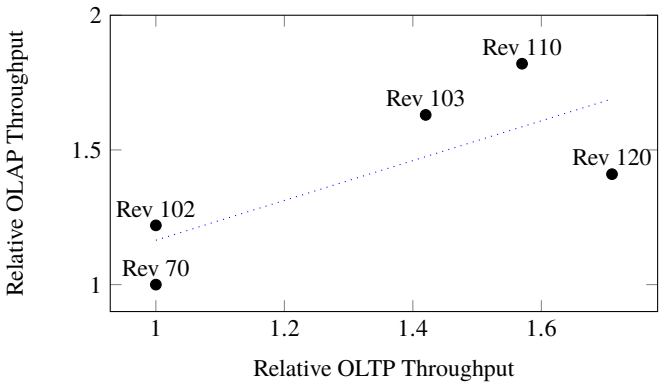


Fig. 7: Evolution of Mixed-workload performance of SAP HANA.

- Pre-aggregation is inflexible because the decision on what needs to be precomputed has to be taken upfront.
- The approach to be taken comes along with high maintenance overhead, scalability issues due to in-place updates, as well as non-negligible storage costs (the tables storing the pre-aggregates can get quite big).

The ability to run mixed workloads on top of SAP HANA allowed to significantly "slim down" application code. By replacing these redundant aggregate tables by compatible view definitions on base tables the application does not need to maintain these views during write transactions. This implies that aggregate values are now computed on-the-fly within transactional-style ERP applications [Pla14]. As a consequence, the workload changed towards a mixture of short-running lookup/update operations and long-running aggregate queries even for running only the transactional operations of an ERP system.

As an optimization, SAP HANA exploits an aggressive caching strategy to improve on the overall scalability of query processing. Using state-of-the-art view matching techniques during query compilation, the cache mechanism may provide full transparency for the application.

SAP HANA supports a static caching strategy which refreshes the cache based on a configurable retention time. This alternative is useful when accessing rather static data or for applications that can be satisfied with data that is a few minutes old. Using the static cache we observe a significantly reduced CPU consumption which ultimately leads to a higher throughput for analytical queries. Even refresh intervals of a few minutes add low overhead for refreshing the static cache. In this scenario, it is important that the application makes the age of the returned data transparent to the user.

As a second alternative SAP HANA supports a dynamic caching strategy which keeps a static base version of a view, calculates the delta with respect to the base version and merges it into the result of a transaction-consistent snapshot [BLT86]. Clearly, this alternative trades fresh data for higher CPU consumption for the query execution.

On the application level an optional declarative specification of cached views allows the application code designer to convey hints of caching opportunities to the database.

4.3 Complex Analytics on Normalized, Technical Database Schemas

The fact that OLTP transactions are executed on the same physical database schema as complex analytic queries also implies that there is no ETL step involved which brings the data into a form that is suitable for reporting purposes. Together with the removal of aggregates maintained by the application this leads to the decision to create a layered architecture of database views [MBBL15]. At the lowest level, these views map tables in the physical schema to the most basic business objects which are meaningful to applications like S/4HANA, e.g. sales orders. Higher-level views build on top of these views and represent higher-level business entities, e.g. a cost center or a quarterly sales report. Analytics performed by business applications like S/4HANA execute seemingly simple SQL queries on this stack of views. Clearly, these reporting queries do not run on star- or snowflake schemas as they are commonly used in data warehouses. Instead, complex reports work on the technical, normalized database schema with, e.g. header-lineitem structures and (optionally) references to multiple auxiliary tables, e.g. for application-specific configuration or texts. Figure 8 visualizes this layered architecture of views - also called Core Data Services (CDS) views - as nodes, and the nesting relationship and associations as edges. In a typical S/4HANA development system in 2015 there are 6,590 CDS views with 9,328 associations which sum up to 355,905 lines of code for the view definitions.

The complexity of these business-oriented database views is notable: After unfolding all referenced views, the CDS view with the highest number of referenced tables referenced 4,598 base tables. Specifically, there are many views with a high complexity, i.e. there are 161 CDS views that reference more than 100 tables. Considering these numbers it is clear that analytical queries on these complex nested views are very challenging to optimize and evaluate in an efficient way. So far, only few research papers consider queries of this complexity, e.g. [DDF+09]. One way to handle such complex queries is to apply caching strategies as described already in Section 4.2. Another challenge is the issue of supportability for such complex scenarios, e.g. how can one characterize complex views, or even analyze performance issues considering that a plan visualization of queries on such a complex view does not even fit on a very large poster? Finally, it is difficult to offer metrics to developers who define these complex nested CDS views so that they can assess the complexity of the views they model. For example, only counting the number of referenced tables might be misleading because this measure does not consider the size of the accessed tables. Other metrics like the usage of complex expressions may also be relevant for this assessment.

4.4 Data Aging

In an in memory database like SAP HANA, the data accessed by SQL statements must reside in memory. By default, full columns are loaded into main memory upon first access and only

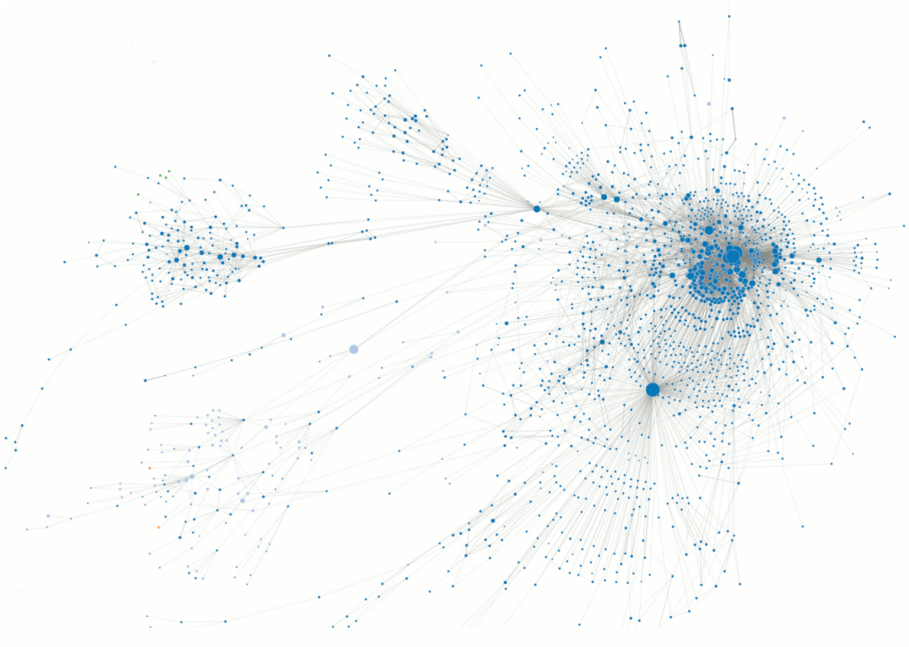


Fig. 8: Relationship of nested CDS views.

memory pressure may force the system to unload columns from memory. Because historical data needs to be kept for a long time, e.g. for legal reasons, keeping all accessed data in memory may often be considered too expensive, especially for rarely accessed, historical data. Other systems, like Hekaton or DB2, offer mechanisms to detect rarely used data and avoid loading it into main memory [CMB+10, AKL13]. However, in our experience, application knowledge that indicates which data is expected to be used rarely is required in order to exploit a maximum of memory footprint reduction while at the same time providing low-latency query response times, and avoiding an accidential placement of data on slow storage media like disk. Consequentially, as part of an *aging run*, the application marks certain data as warm or cold data, and based on these flags, SAP HANA can optimize the storage.

The easiest solution available in SAP HANA are table partitions which are declared to have page-loadable columns [SFA+16]. A more sophisticated approach relies on a completely separate storage engine using dynamic tiering [MLP+15]: Dynamic tiering exposes remote tables stored in the SAP Sybase IQ storage engine as virtual tables inside SAP HANA. This way, SAP HANA can store massive amounts of data and keep it accessible for applications via the efficient disk-based analytic query engine offered by SAP Sybase IQ. This setup has the advantage that applications are completely agnostic to the concrete physical schema design, and all relational data is accessible via one common SQL interface.

Nevertheless, access to the cold data stored on disk should be avoided during query processing. Therefore, SAP HANA relies on 1) partition pruning based on the metadata of

partitioned tables, 2) hints provided by the application that indicate that only hot data is relevant in this particular query, and 3) potentially pruning partitions at runtime based on statistics of accessed table columns and available query parameters.

4.5 Summary

In this section, we presented how SAP HANA can deal with mixed workloads of concurrent OLAP and OLTP statements. The scalability on both dimensions heavily depends on the effective use of memory and CPU resources. Furthermore, interactive analytics on complex hierarchies of views requires techniques like caching but also guidance for developers of analytic applications to analyze the impact of queries on these views. Finally, we discussed how the system supports data aging which keeps cold data stored on disk and loads this data on demand into memory with low overhead.

5 Conclusions and Outlook

In this paper we report on our journey of extending SAP HANA from a system heavily optimized for analytical query processing to a database engine that can also handle transactional workloads such that it can sustain sufficient throughput for large customer systems running SAP ERP workload. As we explain and underline with experiments, this required various specific localized but also architectural changes. As a result, we can report that it is indeed possible to handle OLTP workloads "fast enough" for the majority of workloads we encounter in large customer scenarios, while allowing them to run realtime analytics on the same operational data set.

Nevertheless, we are also convinced that this journey is not over yet. The new opportunities to handle both analytical and transactional workload efficiently without redundancy, complex ETL, or data staleness in a single database results in an increase of truly mixed workload. The demand in this direction goes well beyond what is currently possible - handling mixed workload gracefully with predicable and low response times, high system throughput, and high resource utilization is an ongoing challenge. In addition, while applications become more and more aware of the opportunities of doing analytics, planning, as well as running complex machine learning algorithms on transactional data, we expect that even more data-intensive logic needs to be handled within the database. Finally, as the data sizes grow further, using persistent storage needs to be reconsidered as an integral part for an in-memory database like SAP HANA, e.g. to age warm and cold data to NVM, SSDs or spinning disks.

In summary, the journey of SAP HANA towards a high performance as well as robust foundation of large software applications in combination with demands coming from operating SAP HANA within on-cloud, on-premise as well hybrid deployment settings is not over. Stay tuned!

6 Acknowledgements

We want to thank all members of the global SAP HANA database development team for bringing the system described in this paper from vision to reality. We are also grateful to our colleagues from the ABAP application server development team for the excellent collaboration (e.g. on the FDA and NFAE features), and for providing Figure 8.

References

[AKL13] Karolina Alexiou, Donald Kossmann, and Per-Ake Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB*, 6(14):1714–1725, 2013.

[BLT86] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[Böh15] Alexander Böhm. Novel Optimization Techniques for Modern Database Environments. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 23–24, 2015.

[CMB+10] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Kenneth A Ross, and Christian A Lang. SSD Bufferpool Extensions for Database Systems. *PVLDB*, 3(1-2):1435–1446, 2010.

[DDF+09] Nicolas Dieu, Adrian Dragusanu, Françoise Fabret, François Llirbat, and Eric Simon. 1,000 Tables Under the Form. *PVLDB*, 2(2):1450–1461, August 2009.

[FML+12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[KDR+14] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving in-memory database index performance with Intel® Transactional Synchronization Extensions. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014*, pages 476–487, 2014.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. IEEE ICDE*, pages 195–206, 2011.

[LCC+15] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. Oracle Database In-Memory: A Dual Format In-Memory Database. In *Proc. IEEE ICDE*, pages 1253–1258, 2015.

[LCF+13] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. Enhancements to SQL server column stores. In *Proc. ACM SIGMOD*, pages 1159–1168, 2013.

[Liu11] Huan Liu. A Measurement Study of Server Utilization in Public Clouds. In *Proc. IEEE Int. Conf. on Dependable, Autonomic and Secure Computing*, pages 435–442, 2011.

[LSKN16] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, pages 3:1–3:8, 2016.

[LSP+16] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *ACM SIGMOD*, pages 1307–1318, 2016.

[MBBL15] Norman May, Alexander Böhm, Meinolf Block, and Wolfgang Lehner. Managed Query Processing within the SAP HANA Database Platform. *Datenbank-Spektrum*, 15(2):141–152, 2015.

[McK04] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[MLP+15] Norman May, Wolfgang Lehner, Shahul Hameed P., Nitesh Maheshwari, Carsten Müller, Sudipto Chowdhuri, and Anil K. Goel. SAP HANA - From Relational OLAP Database to Big Data Infrastructure. In *Proc. EDBT*, pages 581–592, 2015.

[MSL+15] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-Efficient Aggregation: Hashing Is Sorting. In *ACM SIGMOD*, pages 1123–1136, 2015.

[Pla09] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proc. ACM SIGMOD*, pages 1–2, 2009.

[Pla14] Hasso Plattner. The Impact of Columnar In-memory Databases on Enterprise Systems: Implications of Eliminating Transaction-maintained Aggregates. *PVLDB*, 7(13):1722–1729, August 2014.

[PSM+15] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *PVLDB*, 8(12):1442–1453, 2015.

[RAB+13] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU Acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[SC05] Michael Stonebraker and Ugur Cetintemel. "One size fits all": an idea whose time has come and gone. In *IEEE ICDE*, pages 2–11, 2005.

[SFA+16] Reza Sherkat, Colin Florendo, Mihnea Andrei, Anil K. Goel, Anisoara Nica, Peter Bumbulis, Ivan Schreter, Günter Radestock, Christian Bensberg, Daniel Booss, and Heiko Gerwens. Page As You Go: Piecewise Columnar Access In SAP HANA. In *ACM SIGMOD*, pages 1295–1306, 2016.

[TMBS12] Frederik Transier, Christian Mathis, Nico Bohnsack, and Kai Stammerjohann. Aggregation in parallel computation environments with shared memory, 2012. US Patent App. 12/978,194.

[WPM+15] Florian Wolf, Iraklis Psaroudakis, Norman May, Anastasia Ailamaki, and Kai-Uwe Sattler. Extending database task schedulers for multi-threaded application code. In *Proc. SSDBM*, pages 25:1–25:12, 2015.

Internet of Things

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 567

Bosch IoT Cloud – Platform for the Internet of Things

Tobias Binz[1]

Abstract

Until 2020 all electronic products of Bosch should be IoT-enabled. This IoT and digital transformation is an enormous opportunity for Bosch, addressing the fields of connected mobility, connected industry (Industry 4.0), connected buildings and smart home. This overarching connectivity strategy is enabled by the Bosch IoT Cloud, a cloud platform specialized on developing, testing, and running scalable IoT services and applications.

[1] Robert Bosch GmbH, Steiermärker Straße 3-5, 70469 Stuttgart-Feuerbach, Tobias.Binz@de.bosch.com

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 569

Industrial Analytics: Methodiken und Datensysteme für das Industrial Internet (IIoT)

Eddie Mönch[1]

Abstract: An einem breiten Spektrum von Kundenbeispielen vom Schiffsdiesel über Kaffeemaschinen bis hin zum Industrieroboter wird aufgezeigt, dass im Bereich Analytics von Industrial Internet of Things (IIoT) Anwendungen der Data Scientist ohne den Domainexperten nicht zu befriedigenden Ergebnissen kommen kann. Die Kundenbeispiele lassen sich im Kern in sechs unterschiedliche Use Cases unterteilen: Root-Cause-Analyse, Fehlerprädiktion, Datenaggregation, Text Mining, visuelle Interaktion und Optimierungsberechnungen über den Gesamtbestand der Maschinen hinweg. Diese werden jeweils anhand von realen Praxisbeispielen aus dem industriellen Umfeld anschaulich demonstriert.

Keywords: Industrial Internet of Things, Analytics, Machine Learning, Big Data, Deep Learning, Cognitive Computing.

1 Einleitung

Der Vortrag zeigt anhand der Praxisbeispiele die Einbettung des Industrial Internet of Things in die Geschäftsstrategie von Firmen, die im Wandel der digitalen Transformation stecken. So genannte smarte Produkte sind mit Sensoren ausgestattete und internetfähige Produkte. Durch ihre Fähigkeiten, Daten zu sammeln, zu analysieren, zu versenden und zu empfangen, werden sie intelligent (smart). Wenn Produkte smart sind, können Unternehmen ihren Kunden Mehrwerte über den eigentlichen Produktnutzen hinaus anbieten und so die digitale Transformation meistern. Es entstehen Smart Services, also zusätzliche Serviceleistungen (Value-added Services), die zahlreiche Anknüpfungspunkte für neue, bisher unbekannte Geschäftsmodelle bieten. Die dabei entstehenden riesigen Datenmengen (Big Data) zu analysieren und Schlüsse daraus abzuleiten, bis hin zur Vorhersage von Ereignissen, stellt die Disziplin von Industrial Analytics dar.

2 Industrial Analytics

Mit Industrial Analytics wird kann der Workflow vom Signal zur Aktion im Service Management System (vgl. Abb. 1) ganzheitlich unterstützt werden, indem Complex Event Processing und Predictive Analytics auf den anfallenden Sensordaten zielgerichtet eingesetzt werden. Dazu ist in der Praxis erforderlich, Wissen aus Service Cases zu extrahieren, um die notwendigen Systemparameter wie Root Causes, Symptome und Lösungen abzuleiten.

Neben Case-Based Reasoning, einem KI-Verfahren, das Cognitive Computing, inspiriert von menschlichem Problemlösen, ermöglicht indem Daten gegen Referenzen auf Zeitreihendaten verglichen werden, wird eine Kombination aus Deep Learning und semantischem Textmining hierzu verwendet. Maschinelles Lernen wird genutzt, um Regeln für das Complex Event Processing abzuleiten. Zur

[1] Empolis Information Management GmbH, SU Industrial Analytics, Europaallee 10, 67657 Kaiserslautern, eddie.moench@empolis.com

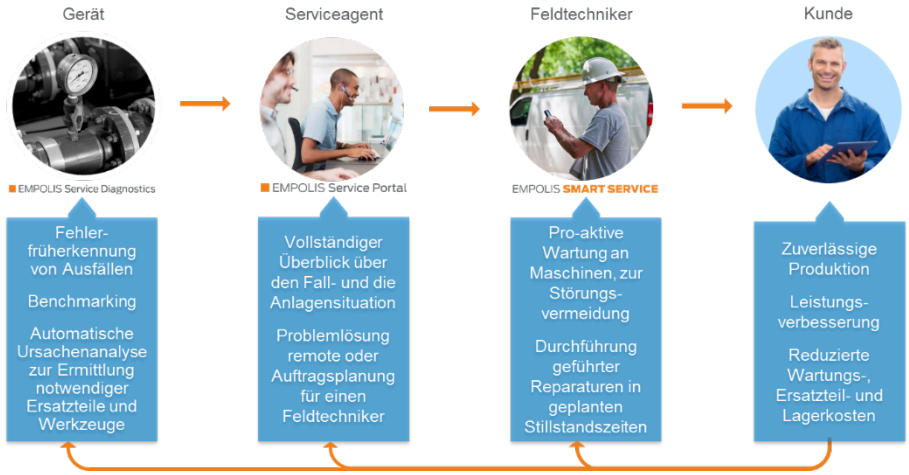


Abb. 1: Smart Services: Vom Signal zur Aktion im Service Management

Abdeckung des gesamten Prozesses gehören dann noch die Integration in Kundensysteme wie Ticketsystem oder CRM und die geführte, im Idealfall automatisierte, nichtlineare Fehlersuche zur Fehlervermeidung bei prädiktiven Alarmen.

2.1 Verfahren im Einzelnen

Die Verfahren des Industrial Analytics werden im Vortrag anhand von Kundenbeispielen vorgestellt und finden sich hier kurz beschrieben wieder:

- **Root-Cause-Analyse:** Bei der Root-Cause-Analyse geht es darum, beim Auftreten einer Störung so schnell wie möglich die Ursache heraus zu finden. Im Vortrag werden Beispiele aufgezeigt, die veranschaulichen, was aus maschinellen Datenströmen gelernt werden kann, um sowohl die relevanten Meldungen von den irrelevanten zu unterscheiden als auch den möglichen Verursacher automatisch heraus zu filtern.
- **Fehlerprävention:** Bei der Fehlerprävention ist das Ziel, Ausfälle vorhersagen und rechtzeitig berechnen zu können, um aus ungeplanten Ausfällen geplante Ausfälle machen zu können, und dem Maschinenbediener oder dem Servicetechniker eine geführte Serviceprozedur anzubieten, damit Fehler gänzlich verhindert werden können. Im Vortrag werden verschiedene Vorgehensweisen und Ansätze dieses Verfahren anhand von realen Kundenbeispielen aufgezeigt.
- **Datenaggregation:** Die Datenaggregation oder Datenverdichtung bietet dem Nutzer die Möglichkeit, das Bild über eine aufkommende oder bereits vorliegende Störsituation zu komplettieren, indem dazu Daten aus verschiedenen Quellen angefragt werden, um sich gegenseitig in einem so genannten Data Lake zu ergänzen. So können Informationslücken in der einen Quelle durch Informationen aus der anderen aufgefüllt werden.
- **Text Mining:** Maschinelle Textanalyse, d. h. das Verstehen von Textinhalten unterschiedlicher Sprachen, ist eine äußerst relevante Informationstechnik im Service. Denn textuelle Daten werden bei Störfällen im Service immer erstellt: die Berichte der Servicetechniker, inklusive Auftrag sowie das jeweilige Ticket oder Service Case. Wir zeigen einige Kundenbeispiele, in denen die Textanalyse eine zentrale Rolle bei der Lösung von Servicefällen spielt.

- **Visuelle Interaktion:** Die visuelle Interaktion mit den generierten Daten aus einer Industrie 4.0 Smart Factory oder einer IIoT-Maschine spielt in allen oben genannten Use Cases eine zentrale Rolle. Die Strukturierung, Aggregation und Visualisierung der Daten ist von zentraler Bedeutung, um bei der Störungsbeseitigung nicht in der Informationsflut zu ertrinken. In den oben angesprochenen Praxisbeispielen wird jeweils auch hierauf eingegangen.
- **Kontinuierliche Optimierungsberechnungen:** Analysen, die Trends in der Masse der Servicevorgänge aufzeigen, sind für die Servicesteuerung, das Qualitätswesen, der Service-Trainingsorganisation und der Entwicklung von immenser Wichtigkeit. So lässt sich beispielsweise aufschlüsseln, in welcher Region der Erde, welches Produkt, unter welchen Einsatzbedingungen, in welchem Zeitraum, welche Probleme zeigt. Dies lässt Rückschlüsse auf die Verteilung und Optimierung der Serviceeinsätze und -organisation, auf die frühzeitige Erkennung von Mängeln in der Garantiephase sowie auf Hinweise für Konstruktions- und Zulieferermängel zu.

3 Lessens Learned aus der Praxis und ein Fazit

Industrial Analytics ist eine komplexe Aufgabe (vgl. Abb. 2), die ein sehr gutes Zusammenspiel verschiedener Disziplinen erfordert. Dabei geht es nicht um Algorithmen, sondern um das Verständnis einer Maschine, seiner Prozesse und darum, dessen Verhalten zu messen und vorherzusagen. Um aussagekräftige, nachhaltige und wirksame Analysen zu erhalten, muss man zudem in verschiedenen Branchen und Ingenieursdisziplinen auskennen, um die Inspiration für großartige Ergebnisse zu erlangen.

Aus Geschäftssicht ist zudem noch weiteres zu beachten: Bei einem Industrial Analytics Vorhaben darf man sich nicht von der Magie und des Reizes der Analytics an sich leiten lassen! Man muss sich auf den Geschäftsnutzen von Analytics konzentrieren, nicht auf Analytics zum Selbstzweck. Viele Projekte scheitern leider an nicht vollständig ausgearbeitetem Business Case.

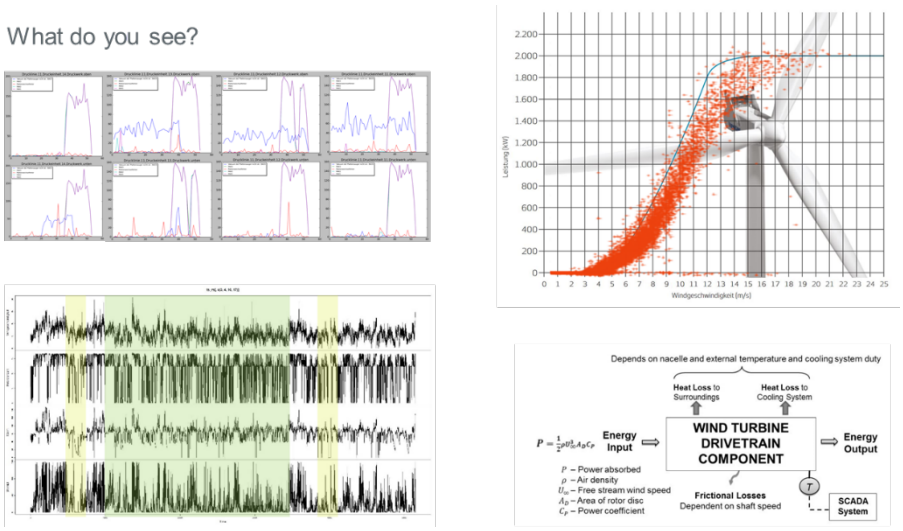


Abb. 2: Klassische Industrial Analytics Aufgabe inkl. ingenieur-seitiges Prozessverständnis

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 573

Designing Test Environments for Cyber-Physical Systems

Harald Vogt[1]

Abstract

The transformation of industrial assets into networked entities promises to increase the productivity of manufacturing processes significantly. It is a prerequisite for the close integration of these assets into production environments that increasingly rely on computerized planning, execution, and monitoring systems. The result of combining a (mechanical) asset with extensive computational capabilities and network resource is called a "Cyber-Physical System" (CPS). In production environments, the quality of these systems is under (at least) the same scrutiny as that of physical-only systems. However, the complexity in achieving this quality level is amplified by the fact that additional, software-induced vulnerabilities are introduced. New approaches to the definition of acceptance criteria, testing strategies, and risk management are required. We describe the quality parameters of CPS, which are intended to guide the design of acceptance criteria. We also describe the challenges in testing CPS and outline the design of a test environment using an example of advanced machine tools.

[1] TRUMPF GmbH + Co. KG, 71252 Ditzingen, Harald.Vogt@de.TRUMPF.com

Technologie und Anwendung

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 577

Bring Your Language to Your Data with EXASOL

Stefan Mandl[1] Oleksandr Kozachuk[2] Jens Graupmann[3]

Abstract: User Defined Functions (UDF) are an important feature of analytical SQL as they allow processing of data right inside of relational queries. Typically UDFs have to be written in a special language which sometimes diminishes their practical use, especially when required libraries are not available in this language. An alternative approach is to allow users to provide functions as native low-level database extensions; an approach that can be very dangerous. EXASOL now follows a radically different approach by allowing to integrate any programming language with the database without affecting data integrity: UDF language implementations are encapsulated within Linux containers that communicate with the database engine via a straightforward protocol. Using these technologies, users can now make available their own programming language for UDFs in the database. As an example, we show how to provide C++ as a UDF language for EXASOL.

Keywords: SQL, User Defined Function, C++

1 Introduction

User Defined Functions (UDFs) are an important tool for users to extend the standard set of SQL functions. Especially for analytical scenarios, the ability to create custom aggregate functions, analytical functions or table generating functions is crucial.

Furthermore, for parallel and distributed systems with massive amounts of data, copying all data to a single machine and processing it there with a single process is often not an option. Here UDFs that run in parallel on many machines provide the necessary solution, but only if the UDF language is suitable for the task and if the right libraries are available in the database.

With EXASOL (starting in version 6.0.0) it is now possible for users or other 3rd party to add new UDF language implementations, new libraries for existing languages, or complete standalone language containers.

In this paper, we explain the basic concepts behind the new technology and demonstrate how to add C++ as a new UDF language to EXASOL.

2 Concepts

Please note, that in EXASOL, UDFs are usually run in parallel and distributed in a cluster. This means, that for a UDF in a given language, there are typically many instances of the

[1] EXASOL AG, Neumeyerstr. 22–26, 90411 Nuremberg, Germany, stefan.mandl@exasol.com
[2] EXASOL AG, Neumeyerstr. 22–26, 90411 Nuremberg, Germany, oleksandr.kozachuk@exasol.com
[3] EXASOL AG, Neumeyerstr. 22–26, 90411 Nuremberg, Germany, jens.graupmann@exasol.com

	scalar output	tuple output
scalar input	scalar functions (SCALAR-RETURNS)	table generating functions (SCALAR-EMITS)
tuple input	aggregate functions (SET-RETURNS)	general $n : m$ mapping (SET-EMITS)

Tab. 1: Different UDF input-output behaviors in EXASOL. The keywords in parentheses are used as syntax in the CREATE SCRIPT statement of EXASOL.

language implementation running at the same time (More information can be found in the white papers [EX15] and [EX]). UDFs can be invoked in two modes:

- Callback mode: Only a single function in the UDF will be called by the database. This mode is used to implement features like virtual schemas, where the SQL compiler needs to communicate with user defined code.
- Compute mode: The UDF is started in order to process data.

In compute mode, data, which is sent from EXASOL to the UDF and results which are sent back from the UDF to EXASOL can either be scalar or tuple valued, resulting in four different kinds of user defined functions which are shown in Tab. 1.

In this section we describe the individual ingredients which together enable users to add new languages to EXASOL:

- Loose coupling of languages to EXASOL gives us a lever where new languages can be added.
- The public script language implementation protocol allows 3rd party developers to create new languages that behave in the same way as the pre-installed language implementations.
- BucketFS allows to actually install the new language on every machine inside an EXASOL cluster.
- A URL like syntax in SQL for describing the details of how to access a script language implementation to enable EXASOL to actually use it.

2.1 Tight vs. loose coupling of UDF languages

In EXASOL language implementations can be one of two different kinds:

- Tightly coupled: The Lua language and its implementation is specifically designed as extension language. Its interpreter is simple and safe and can be integrated with existing C++ code without compromising other parts of the system. Therefore, in order to avoid copying data to and from the script language implementation, Lua is compiled into the database engine.

- Loosely coupled: Implementations of languages like Java, Python, and R are not suited for direct linking into a database engine as they provide means for low-level memory access, tend to clog memory or crash often. These language implementations simply are not compatible with the safety and security requirements of a database management system. Therefore UDF scripts using these languages are started in a separate operating system process which in turn is encapsulated into a separate Linux container. The language implementation and the database system communicate via ZeroMQ[4] sockets.

2.2 Script Language Protocol

The messages between EXASOL and an UDF script language implementation are encoded using Google's Protocol Buffers[5]. The message types are listed in Tab. 2. After being started by the database, the language implementation takes initiative, it sends a message of type MT_CLIENT to the database, which responds with a message of type MT_INFO which among general information about the EXASOL system and the UDF (like database name, database, the number of cluster nodes, etc.) also contains the actual source code of the UDF. Then the script language implementation requests the meta data of the current UDF: types and names of input and output columns, the call mode and the required input and output iteration behaviors.

Finally, the script language implementation notifies the database system that it is up and running and will be requesting and sending actual data using the MT_RUN message.

Then the UDF and EXASOL iteratively interchange data. Here, the UDF requests new data with MT_NEXT messages, and sends results with MT_EMIT messages.

Please note that in an EXASOL cluster a single instance of a UDF typically only processes a small subset of the data. The only exceptions being UDFs with input-output behavior SET-RETURNS and SET-EMITS. For these kinds of UDFs, a single instance will receive the complete data of a group as defined by the GROUP BY clause (if present) of the query it is used in. This allows for implementing aggregate functions by the user. Processing still is in parallel, but the parallelism is follows the grouping of the data.

2.3 BucketFS

BucketFS is a replicated file system in EXASOL clusters which is accessed via HTTP. Files can be stored in buckets which contain no further hierarchy. More than one BucketsFS can be running in an EXASOL cluster. From the point of view of EXASOL's SQL engine, BucketFS is just another external system with one exception: Inside of script language implementations and therefore in UDFs, files in BucketFS can be read via the file system. They are mounted using directory names of the form `/buckets/<bucketfs name>/<bucket name>` As a

[4] `http://zeromq.org`
[5] `https://github.com/google/protobuf`

MT_CLIENT	The script language implementation is alive and requests more information
MT_INFO	Basic information about the EXASOL system and cluster configuration and the UDF script code
MT_META	Names and data types of the data to send between EXASOL and the script language implementation
MT_CLOSE	Terminates the connection to EXASOL
MT_IMPORT	Request the source code of other scripts or information stored in CONNECTION objects
MT_NEXT	Request more data to be sent from EXASOL to the UDF
MT_RESET	Restart the input data iterator to the beginning
MT_EMIT	Send results from the UDF to EXASOL
MT_RUN	Change status to indicate the start of data transfers
MT_DONE	Indicate that the UDF will send no more results for the current group of data
MT_CLEANUP	Send to indicate that no more groups of data will have to be processed by the script language implementation and that it may stop
MT_FINISHED	Sent when the script language implementation successfully stopped
MT_CALL	Used to call a certain function in the UDF when in Single-Call mode
MT_RETURN	Used to send the result of the Single-Call function call

Tab. 2: Message types related to loosely coupled UDF implementations in EXASOL.

matter of fact, the Linux container which is used to run the pre-installed script languages as well as the actual implementation of the pre-installed script languages is also installed in BucketFS.

In a nutshell, this means that languages that are created and installed by 3rd party can use the exact same technology like the pre-installed languages.

2.4 Defining and Using Script Language Implementations in SQL

In order to start and communicate with a script language implementation, EXASOL basically needs two things:

1. A Linux container in which to start the language implementation
2. The actual binary to run inside the container.

These information are customizable via a session/system parameter. The syntax is like this:

```
ALTER SESSION SET SCRIPT_LANGUAGES=
                  '<LANGUAGE_NAME>=<LANGUAGE_DEFINITION> ...';
```

Language names are simple strings while language definitions are either one of the magic names `builtin_python`, `builtin_java`, `builtin_r` or URL syntax. The magic names in turn are interpreted by the EXASOL compiler so that they expand to URL syntax which point to the script languages that come pre-installed with EXASOL. This way, users can be sure that their Python script always uses the Python that comes with the current version of EXASOL, while still being able to modify which implementations to use for other languages.

The URL syntax for script language implementations has the standard form:

```
scheme:[//host[:port]][/]path[?query][#fragment]
```

The parts are interpreted in the following way:

- **scheme** - *localzmq+protobuf* - this means that EXASOL and the script language implementation communicate via ZeroMQ and encode their message with Protocol Buffers. In the future, other schemes could be used.
- **host:port** - currently, this value is empty as script language containers are started on the same machines as EXASOL runs on.
- **path** - this component has the form

  `<bucketfs_name>/<bucket_name>/...<path_to_linux_container>`

  It describes how to access the Linux container which will be started to run the UDF script language implementation in.
- **query** - The value of query is passed as parameter to the actual script language implementation. For the built-in languages, we only have a single binary, which can start all the built-in languages and uses the value of query to determine which language to use.
- **fragment** - this is the path to the binary which implements the language inside the Linux container which is defined in the path component.

For EXASOL 6.0, the pre-installed script language `PYTHON` has the following URL:

```
localzmq+protobuf:///bfsdefault/default/EXAClusterOS/
ScriptLanguages-6.0.0/?lang=python#
buckets/bfsdefault/default/EXASolution-6.0.0/exaudfclient
```

(without newlines).

Once the script language is defined via the variable SCRIPT_LANGUAGES, it can be used when defining new UDFs.

In addition, the language definition can be changed during SQL sessions and UDFs will then use the updated languages.

3 Use Case: Providing C++ as a UDF script language

Up until version 6.0, EXASOL only supported the UDF languages Python, Java, R, and Lua. These are also the languages that come pre-installed with EXASOL 6.0

As an example for the new technology, we show how to add C++ as a UDF language.

3.1 Which Linux container to use?

For new language implementations, user are free to upload new Linux containers into BucketFS. As an alternative they can also use the container which is already provided by EXASOL, which is a good default strategy as it saves disk space.

On the other hand, if your language requires many additional libraries in special versions, a additional container is a good idea.

For implementing C++, we suggest using the container that is already installed in EXASOL. Then only the binary implementing the UDF script language protocol needs to be created by the user.

3.2 Basic strategy for executing C++ in UDFs

As described above, the contents of UDF scripts is sent as source code to the UDF language implementation. Therefore, in order to implement C++, we suggest the following strategy:

- Compile the UDF's code on the fly (using `gcc`).
- Dynamically link and call the generated code.

This strategy has the advantage that it allows to program UDFs in C++ directly inside the SQL client. The disadvantage is that it compiles the C++ code for every query, which in particular for C++ can be very costly.

An alternative strategy is to pre-compile the UDF's C++ code, load it into BucketFS to link and call it from the UDF without invoking the C++ compiler. Then the actual code in the UDF would only specify which libraries to link and which functions to call. As we feel that this is a different (but also interesting) language altogether, we stick with the first alternative.

3.3 Creating the EXASOL Protocol and the Language Implementation

Here comes the hardest part when providing a new language, as the protocol which has been outlined above has to be implemented for every new script language. On the other hand, we provide the source code of the C++ language implementation described here

on `https://github.com/EXASOL/script-languages` which should provide a good starting point for anyone creating new language implementations.

When creating the language implementation, it is important to use the Linux container that later will be used when running the implementation.

Here docker[6] provides a nice tool chain to simplify the process:

- Linux containers can directly be imported from BucketFS and started with shared folders.
- Inside the shared folder, the language implementation can be compiled.
- The content of the shared folder can be packaged and uploaded into BucketFS.

3.4 Creating and Using C++ UDFs in SQL

Once the UDF language implementation is created and loaded into BucketFS, it can be added to the UDF languages known to your EXASOL by adding a definition like the following to the SCRIPT_LANGUAGES variable:

```
CPP=localzmq+protobuf:///bfsdefault/default/EXAClusterOS/
ScriptLanguages-6.0.0#buckets/bfsdefault/cpp/cppclient/cppclient
```

Here we assumed, that `cpplient` implements the UDF language and is stored in a bucket with the name `bfsdefault` and accessible via the path `buckets/bfsdefault/cpp/cppclient/cppclient`.

Now new UDF scripts, using C++, can be created by the user like this:

```
CREATE cpp SCALAR SCRIPT duplicateAndScale(n INT, x DOUBLE,
                                           y DOUBLE, z DOUBLE)
EMITS (x DOUBLE, y DOUBLE, z DOUBLE) AS

%compilerflags -lblas;
#include <cblas.h>

using namespace UDFClient;
```

[6] `https://www.docker.com`

```
void run_cpp(Metadata* meta, InputTable* in, OutputTable* out) {
  for (size_t n = 0; n < in->getInt64(0); ++n) {
    double x[] = {in->getDouble(1), in->getDouble(2),
                                    in->getDouble(3)};
    cblas_dscal(3, 4.323, x, 1);
    for (size_t i = 0; i < 3; ++i)
        out->setDouble(i,x[i]);
    out->next();
  }
}
```

Please note that we added the `%compilerflags` directive to our version of C++ UDFs which can be used for several purposes like linking additional libraries to the C++ code of the UDF. The classes `Metadata`, `InputTable`, `OutputTable` and the convention to use the function name `run_cpp` for the main loop of our UDF are all features of our particular implementation of C++ UDFs and are not related to EXASOL's UDF protocol.

Finally we are able to seamlessly use C++ UDFs in SQL:

```
SQL_EXA> SELECT duplicateAndScale(3,1,2,3);
EXA: SELECT duplicateAndScale(3,1,2,3);

X                Y                Z
---------------- ---------------- ----------------
           4.323 8.646000000000001           12.969
           4.323 8.646000000000001           12.969
           4.323 8.646000000000001           12.969

3 rows in resultset.
```

4 Conclusion

User Defined Functions in EXASOL are a powerful feature that allows processing massive amounts of data in parallel and distributed in a cluster.

They can be used to create map-reduce style dataflows, create new aggregate functions, and support many other use cases.

Except for the language Lua, UDF language implementations are not compiled into EXASOL but run in an isolated Linux container (per UDF instance) for maximum safety and security. These implementations communicate with the database via a straightforward protocol.

By making these these internals accessible for users and other 3rd party, anyone can now add new UDF languages to EXASOL by following the three simple steps of creating a new

language implementation, making it available in the EXASOL cluster via BucketFS and informing the SQL compiler about the new language.

Using these new facilities, a proof-of-concept implementation of C++ as new UDF language has been straightforward and we eagerly expect new languages to be made available by customers, consultants and data geeks.

References

[EX] A Drill-Down into EXASOL. http://info.exasol.com/technical-whitepaper-exasol-2.html.

[EX15] EXASOL – A Peek under the Hood. http://info.exasol.com/technical-whitepaper-exasol-1.html.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 587

Smart Big Data in der industriellen Fertigung

Yvonne Hegenbarth[1], Gerald H. Ristow[2]

Abstract: In der industriellen Fertigung wird ein Bauteil im Laufe des Herstellungsprozesses immer wertvoller, so dass Störungen um Produktionsablauf oder Fertigungsfehler möglichst frühzeitig erkannt werden müssen. Im Zeichen von Industrie 4.0 geschieht dies mittels unterschiedlicher Sensoren, die automatisch ausgelesen werden oder selbst aktiv ihre Werte kommunizieren. Die Daten müssen analysiert und miteinander in Relation gesetzt werden. Es ist wünschenswert, diese Informationen in Echtzeit auszuwerten, insbes. wenn es sich um zeitkritische und aufwändige Herstellungsprozesse handelt. Die Sensordaten können durch Daten aus Logdateien und Datenbanken angereichert werden, um den Maschinenzustand vollständig zu beschreiben. Anhand von konkreten Anwendungsfällen aus der industriellen Fertigung zeigen wir, wie eine echtzeitfähige Streamingplattform helfen kann, die Produktion zu optimieren. Hierbei wird nicht nur die Qualität einzelner Bauteile betrachtet, sondern die gesamte Produktionsanlage, so dass Prozessabweichungen frühzeitig erkannt werden. Ebenfalls werden Vorhersagen sowohl zum Abnutzungsgrad von Maschinen und Werkzeugen gemacht als auch der mögliche Zeitpunkt einer manuellen Intervention vorausbestimmt.

1 Einleitung

Als Firma beteiligen wir uns aktiv an vielen Forschungsprojekten. Das Thema Industrie 4.0 bekommt immer mehr an Bedeutung, wobei die Überwachung und Verbesserung der industriellen Fertigung gefragt ist. Je früher auf Unregelmäßigkeiten oder Störungen reagiert wird, desto geringer ist der Ausschuss, der Maschinenleerlauf oder -stillstand. Insofern ist es erstrebenswert, schon zur Laufzeit bzw. zum aktuellen Produktionszeitpunkt alle benötigten Daten zur Verfügung zu haben. Hier hat sich eine flexible und vielseitig einsetzbare Streaming Analytics Plattform, welche die Daten in (nahezu) Echtzeit verarbeitet, sehr bewährt, siehe z.B. [GS15].

Als Beispiel sei hier das Forschungsprojekt „BigPro – Einsatz von Big Data-Technologien zum Störungsmanagement in der Produktion“[3] genannt, aus dem einige Erkenntnisse und Ergebnisse vorgestellt werden. Konkret sollen zwei Anwendungsfälle aus der Sensorfertigung betrachtet werden:

1. Das Plasmaätzen in der Waferproduktion, auf den detailliert in Abschnitt 2 eingegangen wird. Hierbei wird die Qualität des aktuellen Ätzvorgangs mithilfe von Data Mining Techniken bestimmt, um Ausschuss frühzeitig zu erkennen.
2. Das Testen von Drucksensoren, auf das detailliert in Abschnitt 3 eingegangen wird. An dieser Stelle werden die Sensordaten über das Einlesen von Logdateien gewonnen,

[1] Software AG, Reseach, Uhlandstrasse 12, D-64297 Darmstadt, Yvonne.Hegenbarth@softwareag.com
[2] Software AG, Reseach, Uhlandstrasse 12, D-64297 Darmstadt, Gerald.Ristow@softwareag.com
[3] http://www.fir.rwth-aachen.de/en/research/research-projects/bigpro-01is14011

welche noch synchronisiert werden mussten. Des Weiteren bestimmen wir den Maschinendurchsatz und berechnen daraus, wann ein manuelles Eingreifen nötig sein wird.

In Abschnitt 4 geben wir eine kurze Zusammenfassung und einen Ausblick auf die noch geplanten Aktivitäten im BigPro Forschungsprojekt.

2 Anwendungsfall – Plasmaätzvorgang in der Waferproduktion

Micro-Electro-Mechanical Systems, sogenannte MEMS, sind heute allgegenwärtig, meist als Sensoren oder Aktoren. Diese sind in der Lage Abmessungen im Mikrometerbereich zu messen. Sie bestehen meist aus Silizium und werden aus Wafern gefertigt. Der bekannteste Prozess zur Herstellung ist das Plasmaätzen. Hierbei wird eine Lackmaske aufgetragen, die Bereiche vor dem Ätzangriff schützt. Es entstehen beim Ätzen als Reaktionsprodukte u.a. Polymere, die sich in der Ätzkammer abscheiden. Während des Ätzens wird der Wafer in der Ätzkammer durch die Zufuhr von Helium von der Rückseite gekühlt.

Abb. 1 zeigt den schematischen Aufbau des Helium-Systems, zum Kühlen des Wafers in der Ätzkammer.

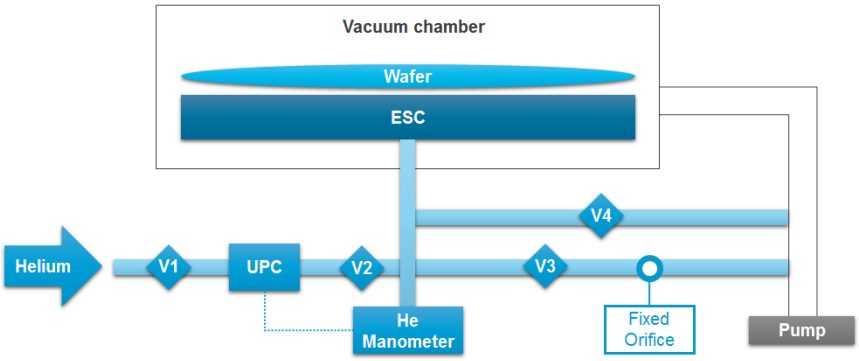


Abb. 1: Schematischer Aufbau der Plasmaätzkammer, aus [He16]

Der Wafer wird auf einem elektrostatischen Chuck (ESC) abgelegt und dort mittels eines elektrischen Feldes fixiert. Eine Pumpe sorgt für ein Vakuum in der Ätzkammer. Über den Unit Pressure Controller (UPC) wird ein Heliumdruck durch den ESC an die Rückseite des Wafers angelegt, der durch das Manometer gemessen wird.

In Gesprächen mit Domainspezialisten wurden mögliche Störungsfälle identifiziert und nach ihrer Häufigkeit und ihrem Einfluss auf den Produktionsablauf klassifiziert. Dabei kam heraus, dass die schwerwiegendste Störung Partikel zwischen Wafer und dem ESC sind, die zu einer schlechten thermischen Ankopplung und dem Austritt von Helium in die Ätzkammer führen. Eine häufige Ursache ist die Verunreinigung durch zu viele abgeplatzte Polymere in der Ätzkammer. Als Kenngrößen wurde die visuell erfasste Polymerfadendicke und der vom Manometer gemessene Heliumfluss identifiziert.

Da z.Zt. noch keine geeignete Schnittstelle der Sensoren zur Verfügung steht, wurden die zu erwarteten Werte in enger Anlehnung an das System simuliert. Abb. 2 zeigt die möglichen Systemzustände, parametrisiert durch den horizontal aufgetragenen Heliumfluss und die vertikal aufgetragene Polymerfadendicke. Es werden vier gut voneinander trennbare Bereiche, sogenannte Cluster, definiert.

i) kein Heliumfluss (ganz links)
ii) sehr hoher Heliumfluss, verursacht durch das Einschwingen (ganz rechts)
iii) kleiner, sehr klar definierter Heliumfluss (links in grün markiert)
iv) hoher, schwankender Heliumfluss für dickere Polymerfäden (rechts in rot markiert)

Der rot markierte Bereich ist das kritische Cluster. Wenn das System sich in diesem Zustand befindet, ist der Wafer höchstwahrscheinlich unbrauchbar. Das System muss daraufhin angehalten und die Ätzkammer von den Polymerfäden gereinigt werden.

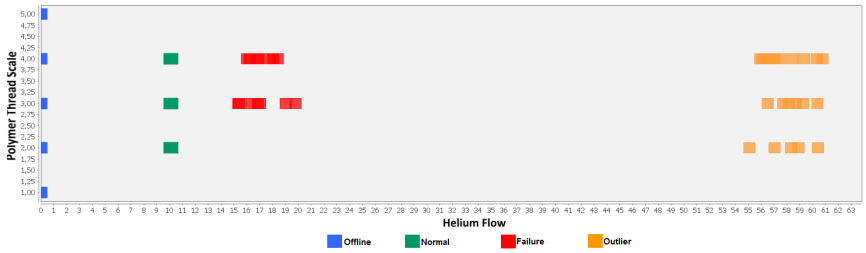


Abb. 2: Visuelle Clusterzuordnung gegeben durch Heliumfluss und Polymerfadendicke, aus [He16]

Im Allgemeinen kann die Einteilung in unterschiedliche Cluster nicht so einfach manuell vorgenommen werden, z.B. wenn der Parameterraum höherdimensional ist, so dass automatisierte Verfahren zur Anwendung kommen. In [He16] wurden verschiedene Clusteranalyse-Algorithmen auf ihre Qualität im hier beschriebenen Anwendungsfall untersucht. Im Rahmen dieser Ausarbeitung werden einige relevante Ergebnisse kurz vorgestellt.

Als Clusteringverfahren wurde u.a. der k-Means Algorithmus, siehe [Ma67] und auch [De14], angewendet. Erst bei einem Output von acht Clustern konnte eine annehmbare Clusterzuordnung erreicht werden. Drei der vier zuvor beschriebenen Bereiche werden sehr gut erkannt. Weitere fünf Cluster beschreiben den Normalbetrieb, welche allerdings zusammengefasst werden können. Eine Vorgabe von weniger Clustern führte in diesem Fall zu einer noch größeren Abweichung des im Abb. 2 beschriebenen Sollzustands. Das Ergebnis des k-Means Algorithmus ist in Abb. 3 gezeigt.

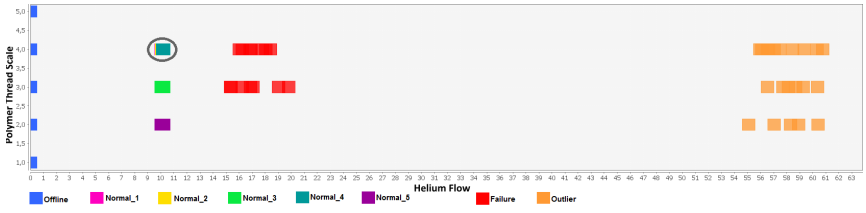


Abb. 3: Streudiagramm k-Means (8 Cluster), aus [He16]

Bei der Verwendung des hierarchischen Clustering Algorithmus werden nur fünf Cluster berechnet. Das Ergebnis wird in Abb. 4 dargestellt. Der Normalbereich wird hier als eine zusammengehörige Klasse identifiziert. In diesem Fall ordnet der Algorithmus den Outlier-Bereich (Abb. 4, ganz rechts) zwei Cluster zu, die wiederum leicht zusammengefasst werden können.

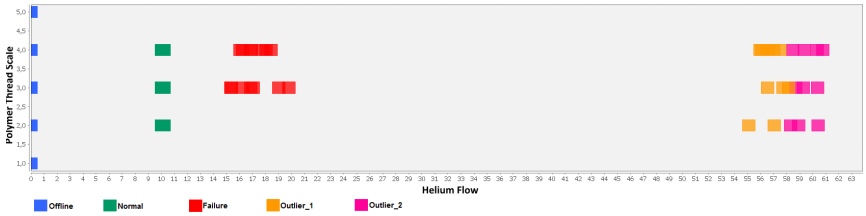


Abb. 4: Streudiagramm hierarchische Clusteranalyse, aus [He16]

Nachdem durch die Clusteranalyse eine Klassenzuweisung mit Testdaten vorgenommen worden ist, kann ein Klassifikationsalgorithmus, wie Naive Bayes oder Decision Tree, ausgewählt werden. Mit Hilfe eines geeigneten Klassifikators wird in [He16] gezeigt, dass eine hohe Vorhersagegenauigkeit der Daten in Hinblick auf ihre Klassenzugehörigkeit erreicht werden kann. Durch das Exportieren und Laden einer Predictive Model Markup Language (PMML) Datei in eine Streaming Analytics Plattform ist eine Vorhersage in nahezu Echtzeit lösbar.

Nach Verfügbarkeit der entsprechenden Schnittstellen zu den Sensordaten, kann mit Zuversicht eine automatische Erkennung der Qualität des Ätzvorgangs und damit der des Wafers durchgeführt werden.

Normalerweise wird in unserem Szenario der aktuelle Maschinenzustand durch die Verknüpfung von unterschiedlichen Ereignissen, sogenannten Events, bestimmt. Die Analyse erfolgt durch Abfragen, sog. Queries, die den Ereignis- und Parameterraum überwachen und auf Auffälligkeiten, sog. Patterns, untersuchen. Da eine Auffälligkeit oder ein Störungsfall durch ein kompliziertes Pattern beschrieben werden könnte, macht es Sinn sich zu überlegen, ob schon ein Teilpattern auf so etwas hindeutet. Es erscheint sinnvoll, die Teilpattern mit Wahrscheinlichkeiten zu versehen, um auch bisher noch nicht aufgetretene Auffälligkeiten zu erkennen und darauf aufmerksam zu machen. Erste Ergebnisse sind sehr vielversprechend und weitere Details sind in [Se16] aufgeführt.

3 Anwendungsfall – Testen von Drucksensoren

Der zweite Anwendungsfall betrifft das automatisierte Testen von Drucksensoren. Dazu werden die Sensoren von Greifern auf Schlitten platziert und durch die Anlage bewegt. In dem hier beschriebenen Anwendungsfall wird der Sensor bei zwei unterschiedlichen Raumtemperaturen, zu jeweils vier verschiedenen Drücken getestet und entsprechend kalibriert.

Die Anlage ist in Abb. 5 skizziert. Die Sensoren werden unten links angeliefert und nach erfolgreichem Test rechts auf eine Rolle bzw. Gurt aufgebracht (5). Defekte Sensoren, die den Testvorgang nicht bestanden haben, werden in einen Abfallkorb platziert (6). Die Schritte sind im Einzelnen:

1. Die Sensoren werden aus dem Vorratsbehälter auf den Schlitten platziert.
2. Der Schlitten gruppiert die Sensoren und wird in die Testvorrichtung gelegt.
3. Die gruppierten Sensoren werden nacheinander in zwei Testglocken gefahren und dort bei unterschiedlichen Temperaturen und Drücken getestet.
4. Die Sensoren werden wieder in Gruppen aus dem Testbereich entfernt und die Testergebnisse werden revidiert.
5. Bei einem positivem Testergebnis wird der Sensor auf eine Rolle bzw. Gurt platziert,
6. hingegen bei einem negativem Testergebnis wird der Sensor als Ausschuss markiert.

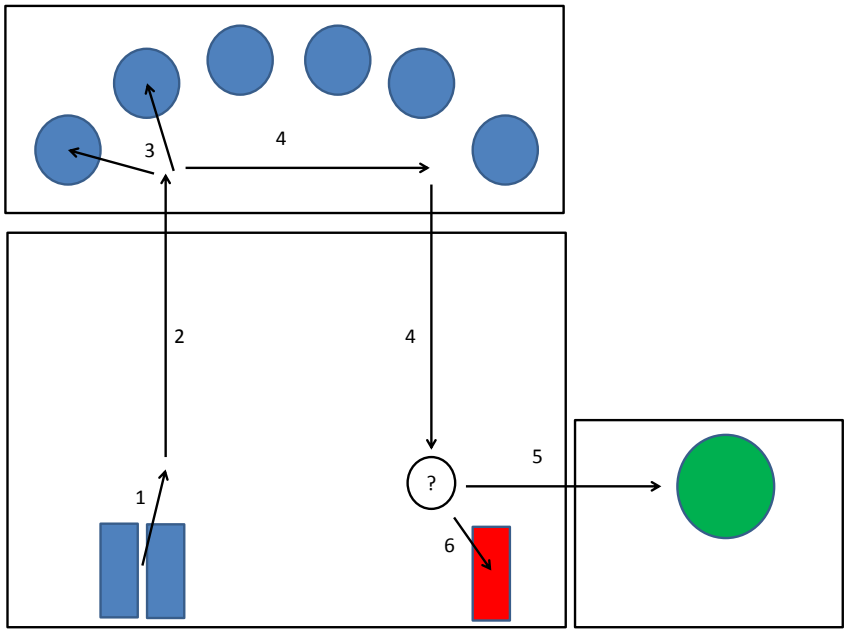


Abb. 5: Skizze der Testanlage mit markiertem Weg des Sensors

3.1 Synchronisation & Echtzeitabfragen der Maschinen-Logdateien

Während dem gesamten Prozess werden sowohl Ereignisse als auch Testergebnisse aufgezeichnet und größtenteils in Logdateien abgelegt. Ein direkter Zugang zu den Maschinen-

bzw. Sensordaten ist noch nicht möglich, jedoch geplant. Aufgrund dessen müssen die Logdateien zeitnah eingelesen und ausgewertet werden, um den aktuellen Zustand der Maschine und seiner Komponenten zu kennen. Am Ende wird jeder Sensor überprüft, ob dieser wirklich auf den Gurt platziert werden soll, oder in den Abfallkorb.

Insgesamt werden im beschriebenen Anwendungsfall fünf unterschiedliche Logdateiformate ausgewertet. Der Einleseprozess wird zwar synchronisiert gestartet und die entsprechende Anfangszeit markiert, jedoch kann dies im Laufe der Auswertung auseinanderlaufen, je nachdem, wie viele Informationen pro Logdateiformat weiterverarbeitet werden müssen. Außerdem ist bei der Auswertung der Daten zu berücksichtigen, dass die Maschinenuhren in der Anlage untereinander nicht synchronisiert sind. Somit müssen Echtzeitabfragen robust formuliert werden. Abfragen, welche Daten aus zwei unterschiedlichen Maschinen-Logdateien enthalten, müssen unabhängig voneinander bestehen und werden erst im Nachhinein zusammengebracht.

In Abschnitt 3.1.1 wird der implementierte Synchronisierungsmechanismus beschrieben, welcher die Synchronisierung der einzelnen Parserthreads garantiert. Abschnitt 3.1.2 beschreibt das Verfahren der Echtzeitabfragen bei fehlender Maschinen-Zeitsynchronisation.

3.1.1 Synchronisation der Maschinen-Logdateien

Damit die Maschinendaten in ihrer korrekten zeitlichen Abfolge verarbeitet werden, wird ein Steuerprogramm (Scheduler) implementiert. Dieser läuft nach dem zeitbasiertem *Earliest Deadline First* Prinzip. Der Scheduler ließt von jeder Maschine eine Task ein und sortiert diese Task-Liste aufsteigend ihrem Fertigstellungstermin (Deadline). Die erste Task in der Liste bekommt demnach die höchste Priorität, da ihre Deadline als erstes zeitnah endet und der Prozess vorweg fertig sein muss.

3.1.2 Echtzeitabfragen bei fehlender Maschinen-Zeitsynchronisation

Um den Verlauf des Sensors in der Anlage zu verfolgen, muss auf die fehlende Maschinen-Zeitsynchronisation geachtet werden. Andernfalls werden Echtzeitabfragen, welche strikt den zeitlichen Verlauf des Prozess folgen, unerwartet ungültig oder es gehen Daten bzw. Informationen verloren. Grund hierfür ist die im Anwendungsfall fehlende Maschinen-Zeitsynchronisation. Die Herausforderung besteht Informationen aus zwei unterschiedlichen Maschinen-Logdateien zu ziehen, welche jedoch von ihrer zeitlichen Folge abhängig sind. Für diesen Fall muss für jede Maschine jeweils eine Echtzeitabfrage formuliert werden, welche zunächst alle notwendigen Informationen sammelt. Erst nachdem beide Abfragen ausgeführt wurden, können die gesammelten Informationen zusammengeführt werden.

3.2 Bestimmung der Maschinenproduktivität

Ein wesentliches Merkmal zum identifizieren des Maschinenstaus ist u.a. den Durchsatz der Maschine zu ermitteln. In einem zuvor benutzerdefinierten Zeitabstand wird berechnet, wie viele Sensoren von der Maschine pro Stunde verarbeitet wurden. Zur Ermittlung der Kennzahl wird in der Streaming Analytics Plattform ein sich stets schiebendes Zeitfenster (Sliding Time Window) definiert. In Abb. 6 wird die Auswertung der Maschinenproduktivität dargestellt.

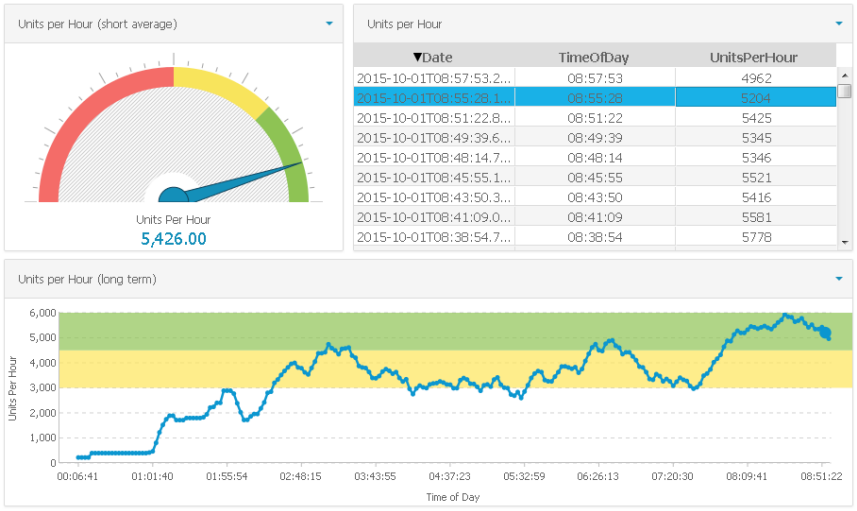


Abb. 6: Maschinenproduktivität einer Testanlage für Drucksensoren

In der Tabelle (oben rechts) werden die aktuell eingetroffenen Daten zur Auswertung der Maschinenproduktivität angezeigt. Das Barometer (oben links) visualisiert den aktuellen Status der Maschine. Bei einem hohen Durchsatz zeigt die Nadel in den grün eingezeichneten Wertebereich. Sinkt der Durchsatz, deutet die Nadel auf den gelben Bereich. Wird der rote Wertebereich angezeigt, läuft im Umkehrschluss die Maschine nicht störungsfrei. Den Verlauf der Maschinenproduktivität in den letzten neun Stunden wird als Liniendiagramm (unten) dargestellt mit ebenfalls eingefärbten Wertebereichen. Zu Beginn der Messung ist ein stärkerer Anstieg anzunehmen, da die Maschine neu gestartet wird. Nach ca. 1-2 Stunden hat sich der Messwert eingependelt und gibt Aufschlüsse über den Zustandsverlauf der Maschine.

3.3 Vorhersage für einen manuellen Eingriff

Ein positiv getesteter Sensor wird auf einen von zwei vorhanden Gurten in der Anlage platziert. Auf einen Gurt passen entweder 5.000 oder 10.000 Sensoren. Ein Mitarbeiter sieht von außen zwar, welche Gurtgröße angebracht wurde, kann aber während der Maschinenlaufzeit nicht genau abschätzen, wann der Gurt voll wird und manuell ausgewechselt werden muss. Dies hat zur Folge, dass im Werk entweder Gurte zu früh ausgewechselt werden oder zu spät und die Maschine still steht. Durch eine *Verhältnisgleichung* (Dreisatz) kann der

Zeitpunkt zum Auswechseln des Gurtes berechnet werden. Zur Berechnung wird hierfür die Gurtgröße benötigt, zusammen mit der aktuellen Anzahl an Sensoren auf dem Gurt und den in Abschnitt 3.2 berechneten Durchsatz pro Stunde. In Abb. 7 wird das Abschwächen der verbleibenden Zeit (in Minuten) für den Wechsel des Gurtes bzw. der Rolle dargestellt.

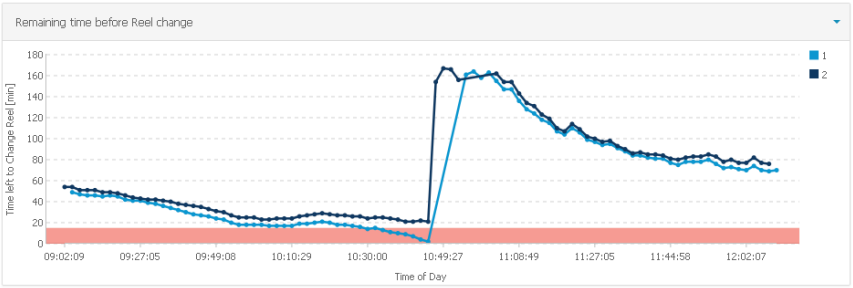


Abb. 7: Automatisierte Vorhersage für die verbleibende Zeit für das Wechsel der Rollen

Als Liniendiagramm sind die zwei Gurte zu erkennen, welche parallel befüllt werden. Auf der y-Achse wird die noch verbleibende Zeit dargestellt, bis die Rolle überfüllt ist. Erreicht die Linie den roten Bereich, soll sich der Mitarbeiter darauf einstellen, dass der Gurt zeitnah ausgewechselt werden muss. Der im Diagramm zu sehende Ausschlag zeigt, dass hier beide Gurte ausgetauscht wurden. Gurt Nr.1 wurde gewechselt, weil dieser tatsächlich voll war, allerdings hätte Gurt Nr.2 erst nach ca. 20 Minuten gewechselt werden müssen.

4 Zusammenfassung und Ausblick

Wir haben anhand zweier Beispiele aus der indiustriellen Chipfertigung gezeigt, wie eine echtzeitfähige Streaming Analytics Plattform[4] benutzt werden kann, um Aussagen über die Stabilität und Qualität des Fertigungsprozesses zu erhalten. Zum einen konnten Aussagen über die Güte von einzelnen Sensoren bestimmt werden, doch auch Aussagen über den Maschinenzustand sind leicht möglich, wenn man das nötige Expertenwissen einfließen läßt. Z.B. konnte der momentane Durchsatz bestimmt werden, oder auch der zeitliche Verlauf, so dass Auffälligkeiten leicht erkennbar sind. Desweiteren haben wir gezeigt, wie man auch Vorhersagen über eintreffende Ereignisse machen kann, so dass die Plattform auch für Predictive Maintenance Aufgaben einsetzbar ist.

5 Danksagungen

Wir danken Roland Schmidt, Suad Sejdovic, Joachim Teutsch, Tim Friedrich und Sebastian Büttner für anregende und fruchtbare Diskussionen und Anmerkungen.

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, und Forschung unter dem Förderkennzeichen 01IS14011C gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

[4] siehe http://www.apamacommunity.com

Literatur

[De14] Dean, J.: Big Data, Data Mining, and Machine Learning. John Wiley & Sons, Inc., Hoboken, New Jersey, 2014.

[GS15] Gärtner, D.; Schimmelpfennig, J.: Vom Sensor zum Geschäftsprozess – Industrie 4.0 in der Stahlindustrie. In (Köhler-Schute, C., Hrsg.): Industrie 4.0: Ein praxisorientierter Ansatz. KS-Energy Verlag, Berlin, S. 128–136, 2015, ISBN: 978-3-945622-01-8.

[He16] Hegenbarth, Y.: Vorhersagemodelle mit Complex Event Processing zur Störungserkennung in einer Chip-Produktion, Bachelorarbeit, Hochschule Darmstadt, 2016.

[Ma67] MacQueen, J. B.: Some Methods for Classification and Analysis of MultiVariate Observations. In (Cam, L.; Neyman, J., Hrsg.): Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability. Bd. 1, University of California Press, S. 281–297, 1967.

[Se16] Sejdovic, S.; Hegenbarth, Y.; Ristow, G. H.; Schmidt, R.: Proactive Disruption Management System: How Not to Be Surprised by Upcoming Situations. In: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems. DEBS '16, ACM, Irvine, California, S. 281–288, 2016, ISBN: 978-1-4503-4021-2.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 597

Fraud Detection 2.0 – Real Time SIP Analytics mithilfe von Complex Event Processing

Markus Schneider[1]

Abstract

Das Ziel des zweijährigen Forschungsprojektes war es, eine Weiterentwicklung der auf dem Markt bereits existierenden Fraudlösungen mithilfe eine CEP-Lösung zu schaffen. Die herkömmlichen Lösungen zeigen Schwächen, weil sie die im Nachgang eines Gespräches aufgezeichneten Abrechnungsdaten (CDR-Call Detail Records) analysieren. Die hier beschriebene Entwicklung analysiert die beim Verbindungsaufbau einer VoIP-Sprachverbindung notwendigen SIP-Signalisierunginformationen (Invites). Somit können Anomalien bereits vor Gesprächsbeginn erkannt werden. Eine nachgelagerte Verarbeitung und Analyse der CDR's birgt immer die Gefahr, dass ein Schaden erst „ex post" erkannt und vermieden werden kann. Insofern redet man hier auch allgemein von einer Schadensminimierung. Eine „ex ante" Missbrauchserkennung mithilfe einer Echtzeitdatenbankanalyse versetzt ein Unternehmen in die Lage, eine Anomalie bereits zu erkennen und demzufolge zu handeln, bevor ein finanzieller Schaden entstehen kann. Die hierbei angewandte Methodik untersucht neben dem Verhalten des Endnutzers hinsichtlich Anomalien während des Verbindungsaufbaus (z.B. Kuba -– diese Destination hat der Endkunde in der Vergangenheit noch nie angewählt), auch eine missbräuchliche Nutzung während eines Gesprächs (Überschreitung von individuellen Thresholds). Anhand der in diesem Projekt eingesetzten Apama Streaming Analytics Platform der Software AG können sehr große Datenmengen in Echtzeit verarbeitet und analysiert werden. Die sogenannten Pattern für die Erkennung der Anomalien wurden von wissenschaftlichen Mitarbeiten der Hochschule Darmstadt in Verbindung mit den Erfahrungen der toplink bezüglich Fraud entwickelt. Die erzielten Ergebnisse wurden in einer Wirkumgebung der toplink GmbH erfolgreich getestet. Diese effiziente Missbrauchserkennung und Vermeidung sorgt nun dafür, dass Telekommunikations-unternehmen dem Thema Fraud zukünftig noch wirksamer entgegnen und somit den finanziellen Schaden für sich und ihre Kunden nochmals deutlich minimieren können.

[1] toplink GmbH, Robert-Bosch-Straße 20, 64293 Darmstadt, markus.schneider@toplink.de

Demo Program

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 601

ControVol Flex: Flexible Schema Evolution for NoSQL Application Development

Florian Haubold[1] Johannes Schildgen[2] Stefanie Scherzinger[3] Stefan Deßloch[4]

Abstract: We demonstrate *ControVol Flex*, an Eclipse plugin for controlled schema evolution in Java applications backed by NoSQL document stores. The sweet spot of our tool are applications that are deployed continuously against the same production data store: Each new release may bring about schema changes that conflict with legacy data already stored in production. The type system internal to the predecessor tool *ControVol* is able to detect common *schema conflicts*, and enables developers to resolve them with the help of object-mapper annotations. Our new tool *ControVol Flex* lets developers choose their schema-migration strategy, whether all legacy data is to be migrated *eagerly* by means of *NotaQL* transformation scripts, or *lazily*, as declared by object-mapper annotations. Our tool is even capable of carrying out both strategies in combination, eagerly migrating data in the background, while lazily migrating data that is meanwhile accessed by the application. From the viewpoint of the application, it remains transparent how legacy data is migrated: Every read access yields an entity that matches the structure that the current application code expects. Our live demo shows how *ControVol Flex* gracefully solves a broad range of common schema-evolution tasks.

Keywords: Schema evolution, NoSQL, NotaQL

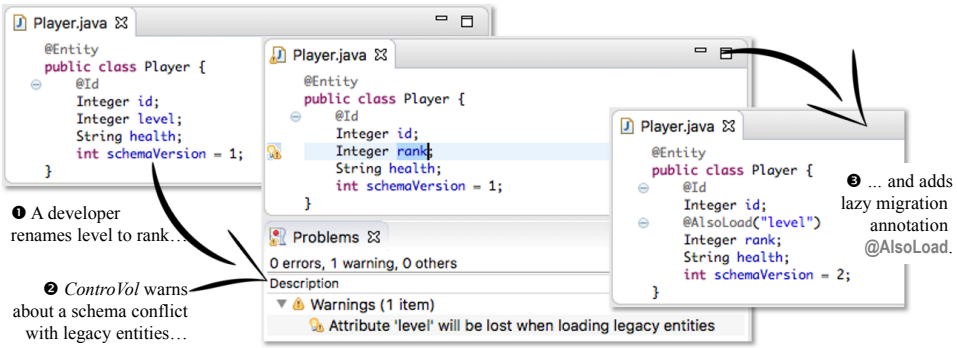
1 Purging Migration Debt in Schema-Flexible NoSQL Data Stores

Schema-flexible NoSQL data stores are popular with agile development teams, especially when software is deployed continuously: Even for small, incremental changes of the code, a new release is deployed to production. Each new version of the application declares its own data model or schema, usually encoded within object mapper class declarations.

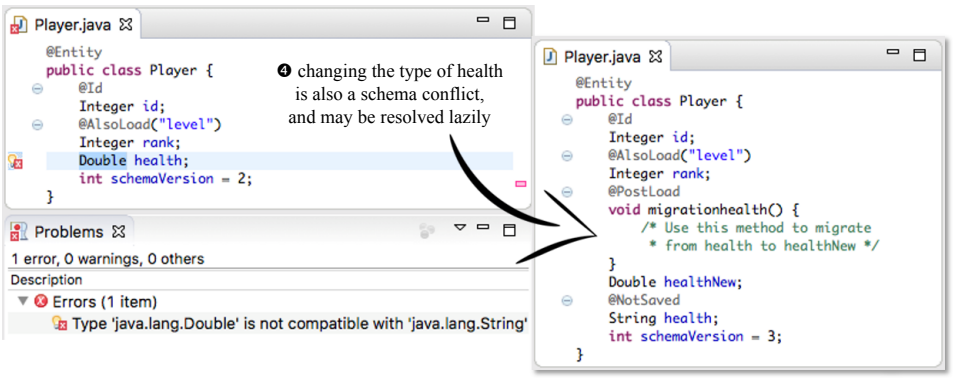
NoSQL data stores like MongoDB [Mon16] can store *legacy* entities, i.e., entities that adhere to the schema imposed by earlier application releases, as well as entities written by the latest release. Object-NoSQL mappers like Morphia [Mor16] are capable of *lazily* migrating legacy entities to the latest schema, whenever they are accessed by the application. Figure 1(a) describes such a scenario for a gaming application: In the first release, each player written to the data store has a unique *id*, and further information on his or her *level* and *health* status. With the second release of the application, the schema of players changes: Attribute *level* is renamed to *rank*. When a legacy player is now loaded, its *level* value is automatically loaded as *rank*, due to the Morphia annotation *@AlsoLoad*.

Simple changes such as adding, removing, or renaming an attribute can be performed quite gracefully with this approach. However, the third release of the application brings about a

[1] Technische Universität Kaiserslautern, f_haubold12@cs.uni-kl.de
[2] Technische Universität Kaiserslautern, schildgen@cs.uni-kl.de
[3] OTH Regensburg, stefanie.scherzinger@oth-regensburg.de
[4] Technische Universität Kaiserslautern,dessloch@cs.uni-kl.de



(a) Lazily renaming *level* to *rank* using the Morphia annotation *@AlsoLoad*.

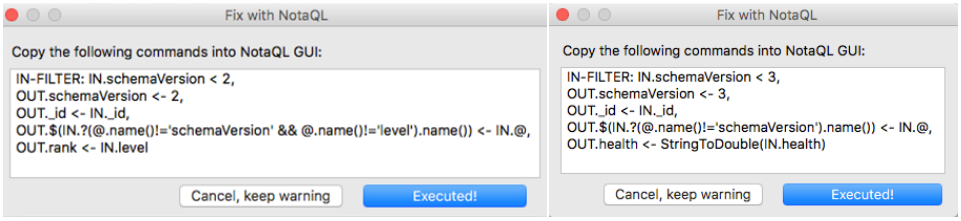


(b) Lazily retyping *health* from String to Double, writing custom code.

Fig. 1: Building up migration debt while lazily evolving the declaration of class Player.

more complex change, as shown in Figure 1(b): The players' health is no longer recorded as a String, but is stored as a floating point value. Lazily retyping values can be done: Whenever a player entity is loaded, the method annotated *@PostLoad* is invoked. Now, developers need to write the code to translate the legacy *health* value (no longer stored due to annotation *@NotSaved*), to a Double (stored as *healthNew*). Already in this simple scenario, we see how quickly we build up technical debt in the form of *migration debt*: Player classes now carry two *health* attributes, to distinguish legacy values from up-to-date values. This can be confusing to newcomers in the project. Moreover, immersing migration code in class declarations violates the software engineering principle of separation of concerns. Additionally, all queries issued by the application code (rather than accessing a single entity by its key) need to consider all structural variations of legacy entities. Overall, application development is slowed down due to the need to account for the structural heterogeneity of legacy entities. At some point in time, *eager* migration of all legacy entities is called for.

Today, developers lack the tool support for systematically managing schema evolution in settings such as these. That is, we need to provide a development environment that



(a) Migrating to schema version 2. (b) Migrating to schema version 3.

Fig. 2: *NotaQL* scripts to eagerly migrate legacy entities, as produced by *ControVol Flex*.

1. keeps track of the various schema versions that occur in the production data store,
2. warns developers about possible schema conflicts when they make changes to class declarations that are incompatible with legacy entities,
3. automatically fixes detected schema conflicts lazily, and further
4. provides easy means so that developers may migrate legacy data eagerly as well.
5. Finally, a tool that even allows to carry out eager and lazy data migration concurrently, which is vital for the continuous deployment of zero-downtime applications.

In earlier work, we have presented *ControVol*, an Eclipse plugin that meets desiderata (1) through (3) [CCS15; SCC15]. In this demo, we introduce its successor *ControVol Flex*, the first tool that meets all five desiderata: *ControVol Flex* generates *NotaQL* [SD15; SLD16] scripts for eager data migration, upon the push of a button. The only requirement that *ControVol Flex* imposes is that all object mapper class declarations carry a dedicated attribute *schemaVersion* (c.f. Figure 1), maintained by *ControVol Flex*. This is a reasonable requirement: Empirical analysis of open source projects shows that maintaining timestamps or versions in persisted entities is common practice in the developer community [RSB16].

Regarding our example, the script in Figure 2(a) transforms all legacy entities written before version 2 of the application code (c.f. line 1) by an update in place: The *NotaQL* commands are read from right-to-left, where the right side matches parts of the input entity (IN), and the left side declares the change to the output entity (OUT). The identifying property *id* (mapped to the MongoDB-internal identifier *_id* by Morphia) is preserved (line 3), as are all properties other than the *level* and *schemaVersion* (lines 4). In fact, the value of a *level*-property is renamed to *rank* (line 5). In line 2, the dedicated property *schemaVersion* is upgraded to 2. Analogously, the *NotaQL* script in Figure 2(b) recasts *health* attributes. By applying both scripts, all legacy entities are eagerly upgraded to schema version 3, and thus the structure expected by the current application code.

A major advantage of *NotaQL* is that this transformation language is independent of a particular data store and even data model: This provides a convenient level of abstraction compared to system-specific APIs or aggregation pipelines. Further, developers may edit the generated *NotaQL* scripts, to unleash the full power of this transformation language in eager migration: *NotaQL* supports complex changes such as nesting and unnesting of hierarchical data, as well as arrays and aggregation operations. As such, it is a powerful tool at the hands of developers for conveniently purging migration debt from NoSQL backends.

2 Demonstration Outline

Our demo scenario describes the agile software-development process of an online role-playing game. The general outline for our interactive demo is this:

1. We introduce a generic development setup using the Eclipse IDE, the Java programming language, the NoSQL data store MongoDB, and the Morphia object mapper.
2. We demonstrate how schema conflicts can occur due to continuous deployment. We provoke serious problems, such as data loss by renaming attributes, type errors by changing attribute types, and missing default values by adding new attributes. *ControVol Flex* detects these conflicts and proposes appropriate quickfixes in Eclipse.
3. We then show how *ControVol Flex* helps to migrate the NoSQL schema lazily by adding Morphia annotations to our code. We also show how *ControVol Flex* generates *NotaQL* scripts to eagerly migrate legacy entities. We point out how user-friendly our plugin is by generating one script for all or even just selected schema conflicts.
4. We demo the two *hybrid modi operandi* of *ControVol Flex*: (1) First kicking off eager migration in the background, while migrating legacy entities lazily, if the application requests access and eager migration has not reached them yet. Alternatively, (2) starting out with lazy migration, and then cleaning up the remaining legacy entities to bring the data instance into a consistent state. We show that application development remains unimpaired by the mode chosen.
5. Furthermore, we demonstrate the automatic version-numbering mechanism for the different stages of our schema evolution process.

Acknowledgements: The authors are grateful to the extended team who has built the predecessor *ControVol*: Eduardo Cunha de Almeida and Pedro Holanda from UFPR Brazil, Thomas Cerqueus from University of Lyon, and Dennis Schmidt from OTH Regensburg.

References

[CCS15] Cerqueus, T.; Cunha de Almeida, E.; Scherzinger, S.: Safely Managing Data Variety in Big Data Software Development. In: Proc. BIGDSE'15. 2015.

[Mon16] MongoDB, http://www.mongodb.org/, 2016.

[Mor16] Morphia, https://github.com/mongodb/morphia/, 2016.

[RSB16] Ringlstetter, A.; Scherzinger, S.; Bissyandé, T. F.: Data Model Evolution using Object-NoSQL Mappers: Folklore or State-of-the-Art? In: Proc. BIGDSE. 2016.

[SCC15] Scherzinger, S.; Cunha de Almeida, E.; Cerqueus, T.: ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development. In: Proc. ICDE'15, demo paper. 2015.

[SD15] Schildgen, J.; Deßloch, S.: NotaQL Is Not a Query Language! It's for Data Transformation on Wide-Column Stores. In: Proc. BICOD'15. 2015.

[SLD16] Schildgen, J.; Lottermann, T.; Deßloch, S.: Cross-system NoSQL data transformations with NotaQL. In: Proc. BeyondMR'16. 2016.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 605

Effective DBMS space management on native Flash

Sergej Hardock[1] Ilia Petrov[2] Robert Gottstein[1] Alejandro Buchmann[1]

Abstract: In this paper we build on our research in data management on native Flash storage. In particular we demonstrate the advantages of intelligent data placement strategies. To effectively manage physical Flash space and organize the data on it, we utilize novel storage structures such as regions and groups. These are coupled to common DBMS logical structures, thus require no extra overhead for the DBA. The experimental results indicate an improvement of transactional throughput for OLTP benchmarks of up to 60% and decrease in write-amplification of up to 2x, which doubles the longevity of Flash SSD. During the demonstration the audience can experience the advantages of the proposed approach on real Flash hardware.

Keywords: Native Flash interface, Flash management, FTL, storage manager, data placement, region.

1 Introduction

To provide backwards compatibility with spinning drives modern Flash SSDs create a black-box abstraction over Flash memory. This is realized inside the device by the so called Flash Translation Layer (FTL). Thus, FTL masks the native properties of Flash memories (e.g. erase-before-overwrite principle, wear-out of Flash blocks, etc.) and emulates the behavior of traditional HDDs, i.e. supporting reads and writes from immutable device addresses. While this architecture makes the replacement of HDDs seamless, it is responsible for underutilizing the performance potential of Flash SSDs. The major disadvantages of the FTL-based black-box SSDs are: (i) DBMS information about data and run-time statistics cannot be utilized to optimize FTL algorithms; (ii) the DBMS neither has control over the physical data placement, nor (iii) over the internal FTL processes, e.g. wear-leveling (WL), garbage collection (GC) or bad-block management (BBM) [CKZ09]; (iv) a high level of functional redundancy along the I/O path down to storage. All these result in high write-amplification (i.e. ratio between the amount of data written by the DBMS and the actually written on physical storage), intensive wear of Flash memory and often unpredictable and state-dependent performance [CKZ09].

To overcome these disadvantages we recently proposed the NoFTL approach [Ha15], which assumes native Flash as secondary DBMS storage. NoFTL removes all intermediate abstraction layers along the critical I/O path (block device interface, file system and FTL), and enables the DBMS to control the physical Flash storage directly. This is achieved by integrating Flash management functionality (address mapping, GC, WL, BBM) into the subsystems of the DBMS (Figure 1). This integration creates the win-win situation for both the DBMS and the Flash management algorithms.

[1] TU Darmstadt, DVS Group, Hochschulstraße 10, 64289 Darmstadt, \protect\protect\T1\textbraceleftlastname\protect\protect\T1\textbraceright@dvs.tu-darmstadt.de
[2] Reutlingen University, DB Lab, Alteburgstraße 150, 72762 Reutlingen, ilia.petrov@reutlingen-university.de

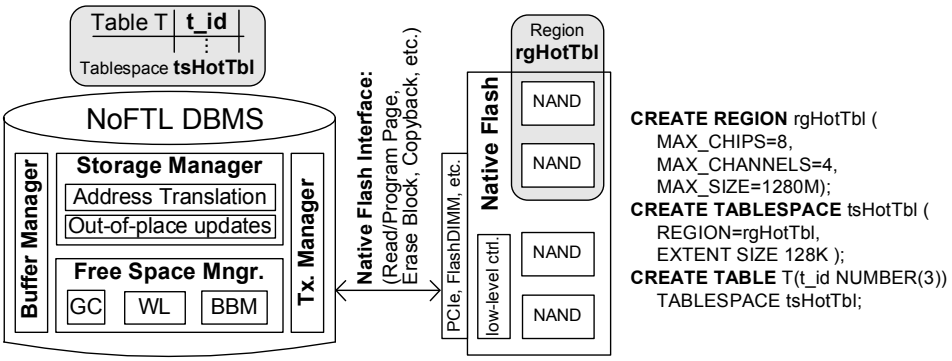


Fig. 1: General NoFTL Architecture including Regions.

In the present paper we revisit traditional methods for physical space management. We introduce two novel storage structures *regions* and *groups*, which allow performing intelligent data placement strategies and reduceing the write-amplification; improve on transaction throughput, and longevity of Flash storage. These structures are coupled to standard logical DBMS structures (tablespaces, extents), hence no extra DBA overhead is incurred.

2 Data organization on native Flash storage

Flash SSDs comprize multiple data channels and Flash chips (dies), which primary define the minimum level of I/O parallelism supported by the device. The general approach in modern SSDs is to spread all the data evenly across the whole physical address space regardless of its properties, since such information is unavailable to traditional black-box SSDs. Although this strategy typically provides good load-balancing and allows for simple WL algorithms, we argue that especially for write-intensive workloads like OLTP it results in (i) enormous write-amplification caused by the GC, (ii) faster wear-out of Flash memory, and (iii) insufficient utilization of available on-device I/O parallelism. To solve these issues we apply an intelligent placement strategies under the NoFTL architecture. NoFTL makes the details about the physical Flash organization visible to the DBMS. This information together with the rich DBMS statistics and metadata provides the basis for efficient data placement. The latter is possible due to the use of native Flash interface and the DBMS control over the physical placement on Flash. To organize and manage data on Flash we introduce a novel storage structure - *regions*. A region comprises multiple data channels and a set of Flash chips. The number of chips in each region as well as the structure of their set is dynamic and can change over time depending on various factors. One or more DB objects with similar access properties can be physically placed in a region; this holds for complete objects or partitions of them. Objects with different properties are placed in different physically separate regions to account and optimize for the specific access characteristics.

The utilization of regions for physical data organization allows for applying hot/cold data separation techniques, which significantly reduce the GC overhead. This in turn, results

in a decrease of the write-amplification and improvement of Flash longevity. Moreover, the proper distribution of data channels and Flash chips among regions, depending on the properties of database objects assigned to them, reduces the contention for physical resources and improves the I/O throughput. Last but not least, the notion of regions introduces only negligible administration overhead for the DBA, since the new structure is coupled to an existing DBMS logical structure - *tablespace*. Consider the example in Figure 1: a region of a certain size *rgHotTbl* is defined over 8 chips. A tablespace *tsHotTbl* is defined on top of *rgHotTbl*, where a newly created table *T* is placed.

In addition to regions we define the notion of *groups*, which are coupled to DBMS *extents*. Groups allow for further hot/cold separation improvement by preventing data with different update frequencies from being mixed within a single Flash block (erase unit). The pages of objects belonging to different groups are kept in separate Flash blocks, thus ensuring higher "temperature homogeneity" within individual blocks. This reduces the write-amplification caused by the GC, by lowering the number of page migrations performed by erasing a victim block [SA13]. Interestingly, the utilization of groups does not harm the even wear-out of Flash blocks within a region, since there is no fixed assignemnt of available blocks to groups. Hence, blocks belonging to different groups are evenly distributed within a certain region, causing thereby its even wear-out in general. Regions and groups exemplify how the utilization of DBMS run-time information on one side, and the knowledge about the Flash physical architecture on the other side, can be used to perform intelligent data placement strategies and achieve higher I/O and transaction throughput, while simultaneously extending the lifetime of Flash SSD. Our experimental results indicate up to 60% improvement in transaction throughput for the TPC-C benchmark and 25% for TPC-B, respectively, with a simultaneous decrease of the performed erase operations by up to 2x, which has a direct impact on the Flash longevity.

3 Demonstration

During the demonstration we introduce the audience to basics of the proposed approach and let them experience it interactively either on real hardware or on a Flash emulator. The demonstration system consists of Flash storage - OpenSSD Flash research board[3] connected to a host PC running Shore-MT[4] (Figure 2). Using an intuitive GUI (Figure 3) the audience can configure a sequence of tests and experience live the performance advantages of the data placement strategies based on utilization of regions and groups. The GUI allows to create multi-region data placement configurations, manage Shore-MT and visually compare the experimental results. The proposed demonstration scenarios are as follows.

Demo-Scenario 1 – Baseline. The audience picks one of the three available OLTP benchmarks (TPC-B, TPC-C or TATP), selects the desired scaling factor (limited by 64GB of Flash storage), the duration of the test and the kind of Flash storage device (Jasmine OpenSSD or Flash Emulator). Shore-MT executes the benchmark using the traditional data placement, i.e. without utilization of regions and groups. During the benchmark run,

[3] http://www.openssd-project.org
[4] https://sites.google.com/site/shoremt/

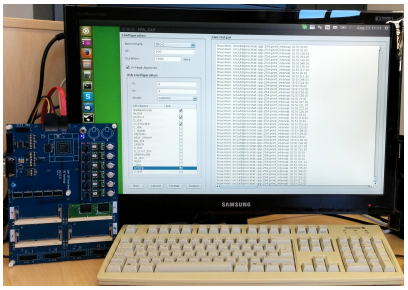


Fig. 2: Demonstration testbed.

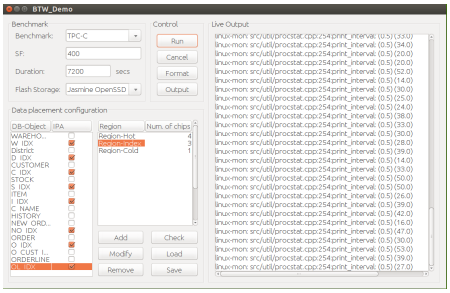


Fig. 3: Demonstration GUI.

the current transactional throughput is visualized. Upon completion, detailed statistics of performed I/Os are shown.

Demo-Scenario 2 – Hot/Cold data separation. In this scenario the audience examines the effects of the data placement strategies based on the hot/cold data separation, which is realized by means of regions and groups. Using the detailed Shore-MT I/O statistics from the *baseline* test the user creates with the help of the demonstration GUI a multi-region data placement configuration. Through the clustering of database objects with similar properties into regions, the overhead of the GC can be significantly reduced. Once the configuration is set and the Flash SSD is completely formatted (low-level) the benchmark is run with the same scaling factor and for the same duration as in the baseline test. The audience can compare the output results of both approaches (throughput, I/O statistics).

Demo-Scenario 3 – Parallelism. This scenario is similar to the previous one, however, the emphasis is placed on controlling the parallelism provided by the Flash storage device. Due to the lack of the SATA NCQ support on the OpenSSD board its level of I/O parallelism is very limited. Thus, the tests in this scenario are performed on the real-time Flash emulator developed in our lab and successfully validated against the real hardware [Ha15]. The Flash chips and data channels of the emulated Flash storage device are divided into regions so that database objects with high demand on I/O concurrency are assigned to regions with more chips and data channels. By doing so the contention for physical resources can be minimized and the available Flash parallelism efficiently utilized.

Acknowledgements. This paper was supported by the German BMBF “Software Campus” (01IS12054) and the German Research Foundation (DFG) project “Flashy-DB”.

References

[CKZ09] Chen, F.; Koufaty, D. A.; Zhang, X.: Understanding intrinsic characteristics and system implications of flash memory based SSDs. In: Proc. SIGMETRICS’09. 2009.

[Ha15] Hardock, S.; Petrov, I.; Gottstein, R.; Buchmann, A.: NoFTL for Real: Databases on Real Native Flash Storage. In: Proc. EDBT’15. 2015.

[SA13] Stoica, R.; Ailamaki, A.: Improving Flash Write Performance by Using Update Frequency. In: Proc. VLDB’13. 2013.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 609

Emma in Action: Deklarative Datenflüsse für Skalierbare Datenanalyse

Eingeladene Demonstration

Alexander Alexandrov[1], Georgi Krastev[1], Bernd Louis[1],
Andreas Salzmann[1], Volker Markl[1]

Abstract: Schnittstellen zur Programmierung paralleler Datenflüsse, die auf Funktionen höherer Ordnung (wie *map* und *reduce*) basieren, sind in den letzten zehn Jahren durch Systeme wie Apache Hadoop, Apache Flink und Apache Spark populär geworden. Im Gegensatz zu SQL werden solche Programmierschnittstellen in Form eingebetteter Domänenspezifischer Sprachen (eDSLs) realisiert. Im Kern jeder eDSL steht ein dedizierter Typ, der verteilte Datenmengen repräsentiert und Berechnungen auf ihnen ermöglicht, wie z.B. DataSet in Flink oder RDD in Spark. Aufgrund der Integration von eDSLs in einer generischen Programmierumgebung (Java, Scala, oder Python) stellen sie eine flexiblere Alternative zu klassischen Ansätzen (z.B. SQL) dar, um gängige Aufgaben (z.B. ETL-Prozesse) in einer skalierbaren, Cloud-basierten Infrastruktur zu implementieren.

Im Laufe der Zeit hat sich jedoch die Anzahl der Operatoren auf den Schnittstellen vergrößert, um Effizienz und Skalierbarkeit zu gewährleisten. Die Schnittstellen werden somit selbst immer komplexer – Ausführungsaspekte wie Join-Reihenfolge, Partitionierung, Caching von Zwischenergebnissen und Verwendung partieller Aggregate, werden dabei nicht vom System, sondern vom Programmierer entschieden. Diese Tendenz hat zwei Nachteile – (1) Programmierer müssen das zugrunde liegende Ausführungsmodell gut beherrschen, um die jeweils passenden Operatoren auswählen zu können, und (2) der erzeugte Quelltext beinhaltet neben den logischen auch physische Aspekte der beschriebenen Berechnung. Dies erhöht die Komplexität von Datenanalyseprogrammen und verringert deren Wartbarkeit und Portabilität.

In dieser Demonstration zeigen wir, wie sich die obengennanten Probleme durch einen besseren Einbettungsansatz vermeiden lassen. Die Grundidee hierbei ist, die Einbettung nicht durch Typen, sondern durch sog. Quasi-Quotes zu realisieren. Diese ermöglichen es, Quelltext-Fragmente in der Host-Sprache auszuzeichnen. Sprachkonstrukte, die durch Typen alleine nicht überladen werden können (wie z.B. Kontrollfluss, anonyme Funktionen, `for`-comprehensions), werden in der eDSL somit reflektiert und in eine strukturell reichere Zwischendarstellung übertragen. Auf dieser Darstellung lassen sich dementsprechend Programmtransformationen definieren, welche die erforderlichen physischen Operatoren und Optimierungen automatisch einführen.

Wir zeigen die oben beschriebenen Ideen in Emma, einer in Scala eingebetteten und auf Quasi-Quotes basierenden DSL. Zur Modellierung verteilter Datenmengen verwenden wir eine algebraische Formalisierung, welche Daten als Multimengen in der sog. UNION Sicht realisiert – d.h. als Instanzen des polymorphen algebraischen Datentypen Bag *A*. Parallele Berechnungen lassen sich dabei mit der Klasse von Programmen, die sich über strukturelle Rekursion definieren lassen, identifizieren.

Anhand von Beispielalgorithmen wie KMeans und NaiveBayes zeigen wir, wie durch Emma komplexe Datenanalyseprogramme ohne besondere Kenntnisse von APIs, lediglich in klassischem Scala, programmiert werden und dann automatisch optimiert und auf massiv-parallelen Big Data Frameworks wie Apache Flink und Apache Spark zur Ausführung gebracht werden. Auf diese Weise leistet Emma einen Beitrag zur deklarativen Spezifikation von Datenanalsyeprogrammen, die im NoSQL-Kontext verloren gegangen war.

[1] TU Berlin, FG DIMA, Einsteinufer 17. 10587 Berlin, name.nachname@tu-berlin.de

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 611

Enabling Efficient Agile Software Development of NoSQL-backed Applications

Uta Störl[1] Daniel Müller[2] Meike Klettke[3] Stefanie Scherzinger[4]

Abstract: NoSQL databases are popular in agile software development, where a frequently changing database schema imposes challenges for the production database. In this demo, we present *Darwin*, a middleware for systematic, tool-based support specifically designed for NoSQL database systems. *Darwin* carries out schema evolution and data migration tasks. To the best of our knowledge, *Darwin* is the first tool of its kind that supports both eager and lazy NoSQL data migration.

Keywords: NoSQL Databases, Schema Management

1 Introduction

In application development, software releases that also change the database schema are a common scenario [Cu08, RSB16]. Due to their schema-flexibility, NoSQL database systems are very popular in such setups, especially with agile software development. However, an outstanding challenge are situations where the structure of data already stored in the production database no longer matches what the latest application code expects. Such *legacy* data must be migrated. Today, many teams rely on custom-coded migration scripts, which is expensive in terms of person hours, as well as error-prone. What is missing is a tool-based support for an optional schema management in NoSQL database systems.

In [KSS14], we discussed the need for a schema-management component capable of managing the schema, schema evolution, as well as data migration within NoSQL database systems. A first prototype called *Darwin*, designed according to these requirements, was sketched in [SKS15]. Now, we present a live demo of a significantly enhanced and extended version of *Darwin*. The key contributions of this *Darwin* demo are:

- *Darwin* supports the complete schema management life cycle: Declaring an initial schema, repeatedly applying schema changes, and migrating legacy data.
- *Darwin* features different options for carrying out data migration tasks. Developers can choose a suitable approach for a given application development scenario.
- *Darwin* is implemented for different types of NoSQL database systems. A public interface makes it easy to integrate further NoSQL database products.

[1] Darmstadt University of Applied Sciences, uta.stoerl@h-da.de
[2] Darmstadt University of Applied Sciences, daniel.n.mueller@stud.h-da.de
[3] University of Rostock, meike.klettke@uni-rostock.de
[4] OTH Regensburg, stefanie.scherzinger@oth-regensburg.de

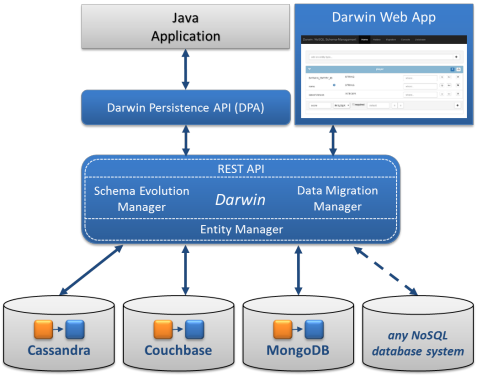


Fig. 1: The *Darwin* system architecture.

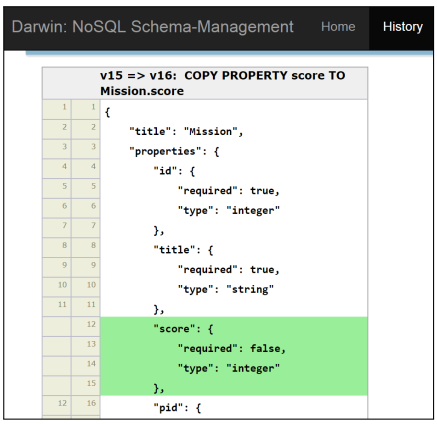


Fig. 2: *Darwin* screenshot: Schema history.

2 Supporting Schema Management with *Darwin*

Darwin operates as middleware between the applications and the NoSQL database systems (see Figure 1). Conceptually, Darwin can support any type of NoSQL database system. Currently, the document stores MongoDB and Couchbase, as well as the column family store Cassandra are supported. Darwin has a system-independent API (c.f. the *Entity Manager* in Figure 1) that makes it easy to integrate other NoSQL systems. We next lay out the schema management process and point out how *Darwin* supports each of its subtasks.

Initial Schema. We assume that an initial schema is available. As illustrated in Figure 3, there are different ways to obtain this schema:

- The schema can be *explicitly* declared by the developers.[5] The *Darwin Web App* (see Figure 1) offers two options, either using the *schema evolution language* introduced in [SSK13], or a *graphical interface*.
- The schema may be *implicitly* derived from Object-NoSQL mappers such as Hibernate or Kundera, or from class declarations within the application code.
- The schema may be *extracted* from data persisted in the NoSQL database in a reverse engineering step. *Darwin* implements schema extraction as proposed in [KSS15].

Schema Evolution. There are different strategies for declaring schema changes (illustrated in Figure 3). *Darwin* uses the schema evolution language first proposed in [SSK13]. Upon request, *Darwin* can also visualize the schema history, as shown in Figure 2.

- The schema changes may be declared *explicitly*. Again, *Darwin* offers two options, using the schema evolution language or a graphical interface. Figure 4 shows an example of a *copy* operation, declared in the graphical interface. The property `score`

[5] An explicitly declared schema is no longer uncommon even for schema-flexible NoSQL databases. For instance, with recent versions of MongoDB, an optional schema may be registered [Mon16].

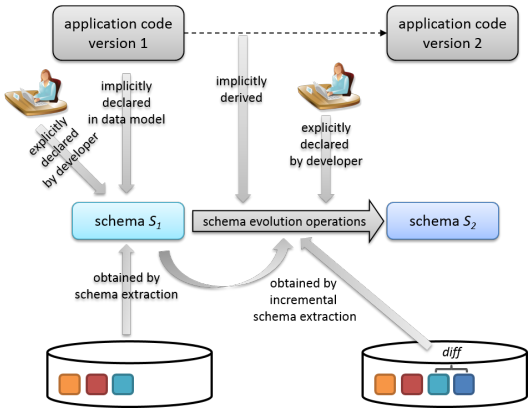


Fig. 3: The schema management process end-to-end.

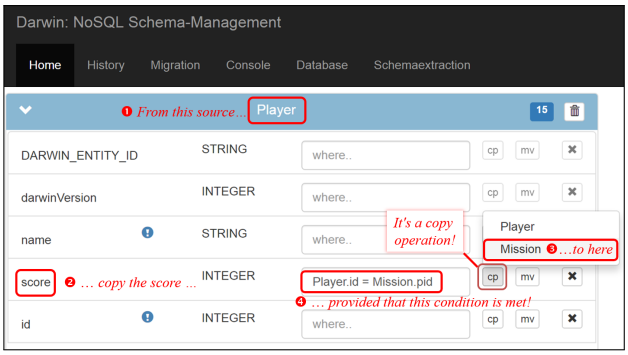


Fig. 4: Declaring a schema evolution operation using the graphical interface of *Darwin*.

is copied from each `Player` to the player's `Mission` entities. *Darwin* then generates the schema evolution operation accordingly:

```
copy Player.score to Mission where Player.id = Mission.pid.
```

- Another way is to *implicitly derive* schema evolution operations by analyzing changes to the application code. Addressing this task is scheduled for future work.
- The third possibility is to *incrementally* maintain a schema: An initial schema is extracted, and then maintained along with all updates to entities in the database. This approach is also implemented within *Darwin*, based on ideas proposed in [La16].

Data Migration. *Darwin* further supports two data migration strategies [KSS14]:

- *Darwin* implements *eager migration*, where the entire legacy data is migrated as a consequence of schema evolution. Eager migration can be explicitly initiated by users of *Darwin*, or it can be declared as the default behaviour of *Darwin*.
- *Darwin* also implements *lazy migration*, where single entities are only migrated on demand, when they are loaded by the application. This mechanism is triggered by calls from the application code to the *Darwin Persistence API*.

3 Demo Outline

Our demo shows how *Darwin* supports the schema management process end-to-end:

1. We start with schema extraction from synthetic gaming data persisted in the database.
2. Visitors of our demo may generate schema evolution operations using the graphical interface (Figure 4) or declare changes in our schema evolution language. We cover adding, removing, and renaming the properties of entities, as well as copying and moving properties between different kinds of entities.
3. Afterwards, we inspect the schema history, as visualized in Figure 2.
4. We can interactively assess the impact of *eager* or *lazy* migration on the data instance.

4 Outlook

Our next step is to implement additional migration strategies besides *eager* and *lazy* migration, to support a broad range of application needs [Kl16]. We further work on detailed cost models for data migration, to help developers choose the most suitable approach.

Acknowledgements: We thank O. Haller, T. Landmann, T. Lehwalder, K. Möchel, H. Nkwinchu, and M. Richter from the Darmstadt University of Applied Sciences for implementation work on *Darwin*.

References

[Cu08] Curino, Carlo; Moon, Hyun Jin; Tanca, Letizia; Zaniolo, Carlo: Schema Evolution in Wikipedia - Toward a Web Information System Benchmark. In: Proc. ICEIS'08. 2008.

[Kl16] Klettke, Meike; Störl, Uta; Shenavai, Manuel; Scherzinger, Stefanie: NoSQL Schema Evolution and Big Data Migration at Scale. In: Proc. SCDM'16. 2016.

[KSS14] Klettke, Meike; Scherzinger, Stefanie; Störl, Uta: Datenbanken ohne Schema? Herausforderungen und Lösungs-Strategien in der agilen Anwendungsentwicklung mit schemaflexiblen NoSQL-Datenbanksystemen. Datenbank-Spektrum, 14(2), 2014.

[KSS15] Klettke, Meike; Störl, Uta; Scherzinger, Stefanie: Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In: Proc. BTW'15. 2015.

[La16] Langner, Jacob: Entwicklung und Bewertung von Verfahren zur inkrementellen Schema-Extraktion aus NoSQL-Datenbanken. Master's thesis, University of Rostock, 2016.

[Mon16] MongoDB, Inc. MongoDB Manual: Document Validation, December 2016. `https://docs.mongodb.com/manual/core/document-validation/`.

[RSB16] Ringlstetter, Andreas; Scherzinger, Stefanie; Bissyandé, Tegawendé F.: Data Model Evolution using Object-NoSQL Mappers: Folklore or State-of-the-Art? In: Proc. BIGDSE'16. 2016.

[SKS15] Störl, Uta; Klettke, Meike; Scherzinger, Stefanie: Kontrolliertes Schema-Evolutionsmanagement für NoSQL-Datenbanksysteme. In: Proc. LWA 2015 Workshops: KDML, FGWM, IR, and FGDB. 2015.

[SSK13] Scherzinger, Stefanie; Störl, Uta; Klettke, Meike: Managing Schema Evolution in NoSQL Data Stores. In: Proc. DBPL'13. 2013.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 615

Energy Elasticity on Heterogeneous Hardware using Adaptive Resource Reconfiguration

(Invited Demonstration)

Annett Ungethüm[1], Thomas Kissinger[1], Willi-Wolfram Mentzel[1], Eric Mier[1], Dirk Habich[1], Wolfgang Lehner[1]

Energy awareness of database systems has emerged as a critical research topic, because energy consumption is becoming a major factor. Recent energy-related hardware developments tend towards offering more and more configuration opportunities for the software to control its own energy-based behavior. Existing research within the DB community so far mainly focused on leveraging this configuration spectrum to identify the most energy-efficient configuration for specific operators or entire queries. In [Un16], we introduced the concept of energy elasticity and proposed the energy-control loop as an implementation of this concept. Energy elasticity refers to the ability of software to behave energy-proportional and energy-efficient at the same time while maintaining a certain quality of service.

Within our demonstration we employ an ODROID-XU3, which is based on the ARM big.LITTLE design. The chip features one cluster of four LITTLE ARM Cortex A7 (frequency range from 200MHz to 1.4 GHz) and one cluster of four big ARM Cortex A15 (frequency range from 200MHz to 2 GHz). Thus, the chip allows 24 active core combinations and 247 frequency combinations (100 MHz steps; shared clock per core cluster) which amounts to a total configuration space of 5,928 exploration points. The ODROID-XU3 also contains four current and voltage sensors. They are used in our demo for individually measuring the power consumption of the A15 and A7 cluster as well as the DRAM.

The demo software itself runs a basic storage subsystem allowing memory-intensive as well as compute-intensive tasks. The user is able to interactively adjust the workload in terms of overall system load, task mix, and desired query latency. Then, our energy-control loop automatically selects one out of 5,928 chip configurations and applies the configuration at runtime to implement our energy-elasticity. Furthermore, the monitoring information is gathered from the system and is visualized in the demo UI.

References

[Un16] Ungethüm, Annett; Kissinger, Thomas; Mentzel, Willi-Wolfram; Habich, Dirk; Lehner, Wolfgang: Energy Elasticity on Heterogeneous Hardware using Adaptive Resource Reconfiguration LIVE. In: SIGMOD. pp. 2173–2176, 2016.

[1] Technische Universität Dresden, Dresden Database Systems Group, 01069 Dresden,
<firstname>.<lastname>@tu-dresden.de

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 617

Exploring Databases via Reverse Engineering Ranking Queries with PALEO

(Invited Demonstration)

Kiril Panev[1] Sebastian Michel[1] Evica Milchevski[1] Koninika Pal[1]

A novel approach to explore databases using ranked lists is demonstrated. Working with ranked lists, capturing the relative performance of entities, is a very intuitive and widely applicable concept. Users can post lists of entities for which explanatory SQL queries and full result lists are returned. By refining the input, the results, or the queries, users can interactively explore the database content. The demonstrated system was previously presented at VLDB 2016 and is centered around our PALEO framework for reverse engineering OLAP-style database queries. How is this useful for exploring data?

Consider a user Alice who needs to make up her mind which smartphone to buy next. Alice is favoring model X, model Y, and model Z, in this order. She is interested in finding explanatory queries and in fact populated rankings that resemble this ranking. PALEO tries to determine such queries who could generate this list and looking at their structure Alice learns about the categorical constraints and ranking criteria used. Given the computed rankings, Alice can further learn about other smartphones that perform perhaps even better, depending on how much PALEO is allowed to deviate from the original input ranking. Assume PALEO returned a ranking of {X, W, Y, Z} with constraints *'storage=16GB'* and *'brand=Samsung'*, ranked by *'battery lifetime'*. What can she learn from that and how can she proceed? She can remove the constraint on the make to get additional offers, she also learned that the model W appears feasible, too. Further, she changes the ranking criteria as *'battery lifetime'* is not the most decisive criterion for her anyways, can distort the ranking slightly to see how generating queries are going to differ, etc.

Developing a system that allows working with rankings in such an exploratory fashion brings up several challenges that need to be addressed. First, subsecond response times are required to allow interactive data exploration. PALEO achieves this by precomputed statistics, decision trees, in-memory processing over a sufficient subset of the data, and low false positive rate in the candidate-query evaluation. Second, the system has to allow approximate answers to the user input, as it is not reasonable to assume that the identical ranking can be retrieved from the database content. We address this, by deeply incorporating distance-measure–based pruning into the candidate query generation. Third, a way to bring candidate queries in an order that reflects a user-perceived notion of interestingness is required. To achieve this, PALEO employs novel insights from mining Web tables corpora to derive a classifier that is able to tell whether or not a non-numerical attribute is semantically meaningful to act as a constraint to the `WHERE` clause of a query.

[1] TU Kaiserslautern, Germany, {panev,michel,milchevski,pal}@cs.uni-kl.de

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 619

InVerDa – The Liquid Database[1]

Kai Herrmann[2], Hannes Voigt[1], Thorsten Seyschab[1], Wolfgang Lehner[1]

Abstract: Multiple applications, which share one common database, will evolve over time by their very nature. Often, former versions need to stay available, so database developers find themselves maintaining co-existing schema versions of multiple applications in multiple versions—usually with handwritten delta code—which is highly error-prone and explains significant costs in software projects. We showcase InVerDa, a tool using the richer semantics of a bidirectional database evolution language to generate all the delta code automatically, easily providing co-existing schema versions within one database. InVerDa automatically decides on an optimized physical database schema serving all schema versions to transparently optimize the performance for the current workload.

Keywords: Database evolution, Database versioning, Co-existing schema versions.

1 Introduction

Current relational database management systems (DBMS) do not support co-existing schema versions within the same database properly. However, this is a very common requirement in today's information systems. While software developers use agile techniques to quickly evolve running applications and frequently publish improved versions, the database is the millstone around the neck. Current DBMSes force developers to migrate a database completely in one haul to a new schema version. Keeping other schema versions alive before and after such a migration typically requires manually written and maintained delta code either in the database (views and triggers) or in the application. Further, finding an optimal physical database schema that serves all co-existing schema versions is notoriously hard, since the mix of mainly accessed schema versions changes over time. With all the described challenges, handling co-existing schema versions within one database is very costly, error-prone, and significantly slows down agile developers of information systems.

This demo[3] showcases InVerDa (Integrated Versioning of Database schemas), which provides a solution to this dilemma. InVerDa uses a declarative Database Evolution Language (DEL) called BiDEL. With BiDEL developers can evolve an existing schema to add a new schema version to a database. InVerDa makes the database instantly available through all **co-existing schema versions** within the DBMS. Data can be read and written through all schema versions; writes in one version are reflected in all other versions. To account for changing workload mixes, InVerDa **transparently optimizes the physical database schema** to continuously ensure a high performance without any further interaction of the developer. Hence, InVerDa greatly simplifies handling co-existing schema versions.

[1] This work is partly funded by the German Research Foundation (DFG) within the RoSI RTG (1907).

[2] Technische Universität Dresden, Database Systems Group, Nöthnitzer Str. 46, 01187 Dresden, Germany
<firstname>.<lastname>@tu-dresden.de

[3] Sneak preview and demonstrator available at `https://wwwdb.inf.tu-dresden.de/research-projects/projects/inverda`

BiDEL—InVerDa's evolution language—provides Schema Modification Operators (SMOs) to create, drop, and rename both tables and columns as well as splitting and merging tables both vertically and horizontally. BiDEL is similar to established DELs [Cu13, He15], with the distinction that its SMOs are specifically designed to be bidirectional. Based on a BiDEL-specified evolution, InVerDa is able to generate all required delta code to make the database instantly available through the new schema version and to propagate data both forward and backward between all schema versions. BiDEL combines standard SMOs for forward evolution with strategies to fill missing information and resolve ambiguity occurring in backward evolution, essentially by adding to each SMO the arguments of its inverse SMO. Since many SMOs are not information-preserving, InVerDa manages additional auxiliary tables to keep the otherwise lost information. We have formally validated the bidirectionality of BiDEL's SMOs by showing that payload data from any schema version N survives a round trip to version $N + 1$ and back to N unchanged and that the same holds vice versa. Hence, each schema version appears like a full-fledged single-schema database.

As applications evolve over time, the user's behavior—and hence the actual workload mix—changes as well. InVerDa continuously monitors the workload: after significant changes in the workload, InVerDa heuristically determines an optimized physical database schema, migrates the data accordingly, and adapts all involved delta code to keep all schema versions accessible. This automatic migration happens transparently to the users and developers without any required interaction. The physical database schema materializes a subset of all tables in all versions. InVerDa ensures that a specific table can be either accessed locally (iff the table version is physically stored) or by propagating the data access through SMOs to other physically stored table versions. Due to the guaranteed bidirectionality of BiDEL's SMOs, we can be sure that no data will be lost, regardless of the physical database schema.

The contributions of InVerDa are the (1) bidirectional DEL BiDEL, (2) automatic generation of co-existing schema versions, and (3) transparent migration to an optimized physical database schema—all on display in the demo. In the remainder, we introduce InVerDa using a comprehensible example (Section 2) and outline demo details (Section 3).

2 InVerDa – User Perspective

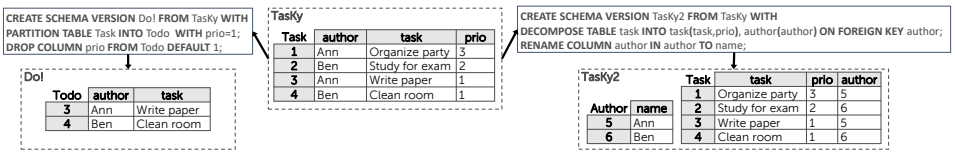


Fig. 1: Example.

We illustrate InVerDa's co-existing schema version support, using the example of a simple task management system called TasKy (cf. Figure 1). TasKy runs as a desktop application backed by a central database. Users can create new tasks and list, update, or delete them. Each task has an author and a priority with 1 being the most urgent priority. In the first release, TasKy stores all its data in a single table `Task(author,task,prio)`.

Development Time As TasKy gets widely accepted, we extend it by a third party phone app called Do! to list the most urgent tasks. Do! uses a different database schema than TasKy. The Do! schema consists of a table `Todo(author,task)` that contains merely tasks of priority 1. Obviously, the initial schema version needs to stay alive for the broadly installed TasKy. InVerDa greatly simplifies this job as it generates all the necessary delta code automatically. The developer simply executes the BiDEL script for Do! (cf. Figure 1), which instructs InVerDa to derive schema Do! from schema TasKy by creating a horizontal partition of `Task` with `prio=1` and dropping the priority column. Executing the script creates a new schema including the view `Todo` with delta code for propagating data changes. When a user adds a new entry into `Todo`, this will automatically insert a corresponding task with priority 1 to `Task` in TasKy. Equally, updates and deletes are propagated to the TasKy schema as well. In sum, the TasKy data is immediately available to be read and written through the newly incorporated Do! app by simply executing the three lines of BiDEL script.

Over time, the TasKy application is further refined and improved. For the next release, called TasKy2, it is decided to normalize the table `Task` into `Task` and `Author`. For an incremental roll-out of TasKy2, the old version TasKy has to remain functional until all clients are updated. Again, InVerDa does the job. When executing the BiDEL script as shown in Figure 1, InVerDa creates the schema version TasKy2 and decomposes the table version `Task` to separate the tasks from their authors while creating a foreign key to maintain the dependency. Additionally, the column `author` is renamed to `name`. InVerDa generates delta code to make the TasKy2 schema immediately available. Write operations to any of the three schema versions are now propagated to all other schema versions.

Operating Time The physical tables initially used for storing the data are the unevolved table versions. All other table versions are implemented with the help of delta code. The delta code introduces an overhead on read and write accesses to new schema versions. The more SMOs are between schema versions, the more delta code is involved and the higher is the overhead. In the case of the task management system, the schema versions TasKy2 and Do! require delta code to propagate data accesses to the physical table `Task`. Assume, some weeks after releasing TasKy2 the majority of the users has upgraded to the new version and heavily uses the mobile phone app, so that TasKy is still accessed but merely by a minority of users. Hence, it is appropriate to migrate data physically e.g. to the table versions of the TasKy2 schema and potentially replicating the data also for Do!'s `Todo`.

Traditionally, the database administrator decides on such a new physical database schema and developers would have to write a migration script, which moves the data and implements new delta code. All that can accumulate to some hundred lines of code, which need to be tested intensively in order to prevent them from messing up the data. InVerDa automatically optimizes the physical database schema and transparently runs the data migration, maintaining transaction guarantees, and updates the involved delta code of all schema versions. No developers need to be involved. All schema versions stay available; read and write operations are merely propagated to different physical tables. In sum, InVerDa allows users to continuously use all schema versions and developers to continuously develop the applications without caring about the physical database schema for a single second.

3 InVerDa – Demonstrator

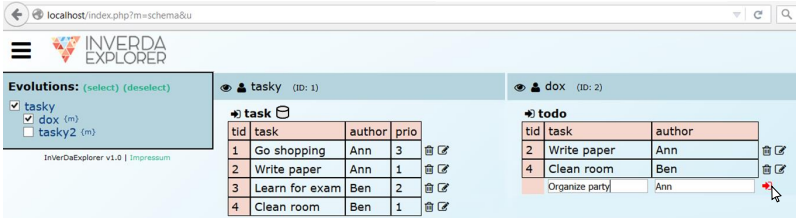


Fig. 2: InVerDa Explorer with TasKy and Do!.

For the sake of demonstration, InVerDa consist of three parts: (1) the InVerDa Console, an Eclipse Plugin to write and execute BiDEL evolution scripts, (2) the InVerDa Explorer, an application for conveniently browsing all **co-existing schema versions**, and (3) the InVerDa Optimizer, a tool running in the background to continuously monitor the workload and **automatically optimize the physical database schema** when necessary. These components operate on top of a PostgreSQL database with common SQL statements. Figure 2 shows the InVerDa Explorer with the versions TasKy and Do!; Task from TasKy is physically stored as indicated by the small cylinder. The InVerDa Explorer allows to manipulate data in any schema version and observe the effects in the other versions. Inserting a task in Do! as shown, also adds the task in TasKy.

During the demo, we will present the prepared TasKy example. However, InVerDa is a working system and we encourage participants to test it interactively with own scenarios. We will execute BiDEL evolution scripts with the InVerDa Console and demonstrate the genuinely co-existing schema versions by manipulating data with the InVerDa Explorer. Participants can experience both the developer perspective by writing own BiDEL scripts, and the user perspective by probing the generated delta code. They can also have a look behind the scenes at the generated delta code. Further, we have prepared dynamic workloads to show how InVerDa automatically adapts the physical database schema. We want to stimulate discussions with practitioners about e.g. use cases and missing aspects as well as with researchers about e.g. technical details and future research questions.

In sum, InVerDa allows multiple applications in multiple versions to share one common database, each having an individual view on the data, while the physical database schema is transparently adapted to a changing workload in the background. Using the richer semantics of a DEL, InVerDa unburdens database administrators and developers from the costly and error-prone tasks of managing co-existing schema versions manually.

References

[Cu13] Curino, Carlo; Moon, Hyun Jin; Deutsch, Alin; Zaniolo, Carlo: Automating the database schema evolution process. VLDB Journal, 22(1):73–98, 2013.

[He15] Herrmann, Kai; Voigt, Hannes; Behrend, Andreas; Lehner, Wolfgang: CoDEL - A Relationally Complete Language for Database Evolution. In: ADBIS 2015, Poitiers, France. volume 9282 of LNCS. Springer, pp. 63–76, 2015.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 623

Invest Once, Save a Million Times – LLVM-based Expression Compilation in PostgreSQL

(Invited Demonstration)

Dennis Butterstein[1], Torsten Grust[1]

We demonstrate how a surgical change to PostgreSQL's evaluation strategy for SQL expressions can have a noticeable impact on overall query runtime performance. (This is an abridged version of [BG16], originally demonstrated at VLDB 2016.)

The evaluation of scalar or Boolean expressions often takes the backseat in a discussion of query processing although table scans, filters, aggregates, projections, and even joins depend on it. SQL expression evaluation in vanilla PostgreSQL is based on the database kernel's family of `Exec`$\cdots$ functions that form an interpreter. During query runtime, this interpreter repeatedly walks expression trees whose inner nodes represent SQL operators while leaves carry literals or table column references. This evaluation strategy has long been identified as CPU-intensive and outright wasteful on modern computing and memory architectures, leading to frequent pipeline flushes and instruction cache pollution. The resulting interpretation overhead is significant and may dominate all other tasks of the query processor: PostgreSQL indeed spends the *lion share of execution time* on expression computation when it processes selected TPC-H queries.

The present work is an exploration of how PostgreSQL can benefit if we **trade SQL expression interpretation for compilation** and thus turn repeated run time effort into a one-time query compile time task. Cornerstones of the approach are:

- The PostgreSQL query optimizer itself remains unchanged—expressions are compiled after planning and just before query execution starts. We adopt a non-invasive approach that—outside of expression evaluation—retains PostgreSQL's Volcano-style pipeline query processor. Compiled and interpreted expression evaluation coexist; both can contribute to the execution of the same query.
- The just-in-time compilation of expressions is based on the LLVM compiler infrastructure which comes in shape of a library that we link with the original PostgreSQL code—LLVM offers high-quality code generation at low compilation times.
- We invest in code size to save runtime effort—a tradeoff that the SQL expression context admits but traditional compilers for programming languages would shy away from.

The resulting runtime gains are made tangible in terms of an interactive demonstration setup that allows cursory as well as deeper looks under the hood of PostgreSQL 9, both pre- and post-surgery.

[1] Universität Tübingen, Department of Computer Science, Database Systems Research Group, Tübingen, Germany, ⟨`firstname`⟩.⟨`lastname`⟩`@uni-tuebingen.de`

References

[BG16] Butterstein, D.; Grust, T.: Precision Performance Surgery for PostgreSQL—LLVM-Based Expression Compilation, Just in Time. *Proceedings of the VLDB Endowment* 9/13, 2016.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 625

Secure Cryptographic Deletion in the Swift Object Store

Tim Waizenegger[1]

Abstract: The secure deletion of data is of increasing importance to individuals, corporations as well as governments. Recent data breaches as well as advances in laws and regulations show that secure deletion is becoming a requirement in many areas. However, this requirement is rarely considered in today's cloud storage services. The reason is that the established processes for secure deletion of on-site storage are not applicable to cloud storage services. Cryptographic deletion is a suitable candidate for these services, but a research gap still exists in applying cryptographic deletion to large cloud storage services. For these reasons, we demonstrate a working prototype for a secure-deletion enabled cloud storage service with the following two main contributions: A model for offering high value service without full plain-text access to the provider, as well as secure deletion of data through cryptography.

Keywords: secure data deletion, cryptographic deletion, data erasure, records management, retention management, key management, data shredding

1 Background

Cloud based storage solutions are a popular service today especially among consumers. They are used for synchronizing data across devices, for backup and archiving purposes, and for enabling access at any time from anywhere. But the adoption of such storage services still faces many challenges in the government and enterprise sector. The customers, as well as the vendors, have a desire to move these systems, or parts of these systems, to cloud environments in order to reduce cost and improve the service. But security issues often prevent customers from adopting cloud storage services.

Cloud storage providers address these issues by offering data encryption in various configurations. The main difference in their implementation of data encryption is the management of encryption keys, and especially the authority over master keys. Cloud providers generally prefer an encryption system where they keep the master keys and have access to unencrypted data. This enables the providers to operate on the customers' data in order to offer advanced services like indexing/searching the data or analyzing it. Generally, a provider with access to unencrypted data is able to provide a higher quality and more useful service to the customer. Providers that allow client-side encryption and do not store any master keys do not have access to the data. They can only operate as data-dumps and offer very limited features to the customer, which makes this an unpopular business model.

Another security aspect that is especially important to government and enterprise customers is the secure removal of deleted data, i.e. disposing of data after its lifetime has passed. Recent

[1] University of Stuttgart, Institute for Parallel and Distributed Systems, Applications of Parallel and Distributed Systems, waizentm@ipvs.uni-stuttgart.de

data breaches have shown that consumer data which was assumed to be deleted could be restored by attackers, causing privacy issues [Os15]. Enterprises and especially governments are bound by a myriad of regulations for information life cycle governance [EP16]. These state specific requirements for how contracts, reports, personal information, and others have to be stored, and when and how they have to be deleted.

Individuals, corporations as well as governments have a requirement for secure data deletion for these reasons. Not only because they want to be compliant with the law, but also in order to avoid storing compromising information when they do not have to.

Today, secure deletion is usually done by physically destroying storage media in the enterprise and government sector. Individuals mostly rely on "virtual shredding" (i.e. overwriting storage blocks). In a cloud storage scenario, both methods are no longer applicable. Physical disks are shared among different applications and even customers. Providers use complex tiered-storage setups or outsource the physical storage themselves. Identifying the physical disks that need to be destroyed, or the blocks that need to be overwritten, becomes difficult to impossible [DW10].

For these reasons, we propose to use data encryption in order to provide secure deletion; i.e. apply cryptographic deletion to cloud storage services. We achieve this by using a key-management based on our Key-Cascade approach [Ba16, Wa17]. We further propose the separation of data and metadata in cloud storage services. This provides the opportunity to encrypt the data client-side and only allow the provider access to metadata. This should incentivize more providers to offer client-side encryption because they can still offer advances services on the metadata.

In this demo, we present a cloud storage system with the two main contributions:

1. Separation of data and metadata to allow the provider to access unencrypted metadata for enabling advances services.
2. Data encryption with a key management that enables cryptographic deletion.

2 MCM Functionality and Architecture

In order to benchmark our key-management mechanism and evaluate the data/metadata separation, we built the Micro Content Management system MCM[2] which will be presented in this Demo.

MCM is based on Enterprise Content Management systems like Box[3]. It stores objects and files inside storage containers in the Swift[4] object store. Whole containers can be transparently encrypted with a key-management mechanism that allows secure deletion of individual objects. Our user interface allows uploading and retrieving files, setting retention

[2] https://github.com/timwaizenegger/mcm-sdos
[3] https://www.box.com
[4] http://docs.openstack.org/developer/swift/

dates and scheduling deletion, extracting and viewing metadata, and analyzing and graphing analyses on this metadata. It also features interactive visualizations of the underlying key management data structures.

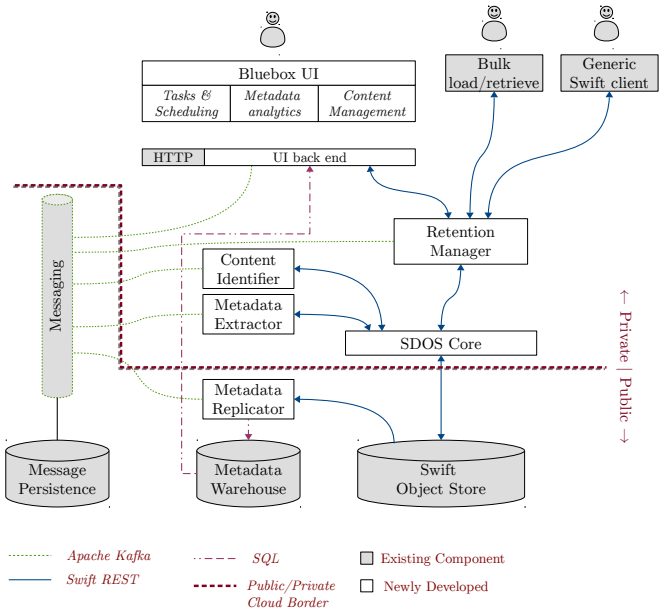


Fig. 1: The Architecture of the Micro Content Management System.

Figure 1 shows the high level architecture of MCM. We use three data management systems (bottom row of Figure 1): An Apache Kafka streaming platform for loosely coupled communication, an SQL database for storing and analyzing the unencrypted metadata, and a Swift object store that holds all the (encrypted) data objects. In order to interface the encryption as well as retention management components with the Swift object store, we designed these services as API proxies for the Swift REST protocol. This enables us to use any unmodified Swift backend (e.g. SaaS) as well as any existing Swift clients. The SDOS encryption and retention manager form a flexible pipeline. All MCM components can run multithreaded or distributed to enable horizontal scaling and high availability. The Kafka streaming platform is used for triggering the execution of jobs for metadata extraction and replication as well as scheduled deletion of old objects. We use a separate metadata warehouse, as Swift lacks advanced querying capabilities for metadata (only retrieving and listing is possible). All the object metadata is primarily stored in Swift and then replicated to the warehouse for analysis. This warehouse is implemented using a relational DBMS because i) the metadata schema is known from the extraction phase and fits well with the relational model, and ii) the intended analytical queries can easily be expressed in SQL.

The location where the components from Figure 1 run is critical to the security of the system. In order to guarantee the secure deletion property, the content of the stored objects must never leave a trusted environment in unencrypted form. The same must be guaranteed for the encryption keys. Our SDOS encryption uses a tree structure for key management of

which only the root key must be kept secure. All other keys are stored encrypted on Swift together with the data objects.

One possible separation of trusted/untrusted environment is given by the red line in Figure 1. It shows that all the data storage system can be outsourced to the public (untrusted) environment, because all sensitive data is encrypted. The metadata replicator only copies data between Swift and the database, so it can run on a public cloud as well. The other components handle sensitive, unencrypted data and the SDOS component has to manage the root key, so these components need to run in a trusted environment. An enterprise cloud gateway appliance could host these sensitive components on premise. Enterprise cloud gateways are already used today for storage outsourcing, they could be equipped with our approach to provide secure deletion as well.

3 Demo Overview

This Demo will show the theory behind cryptographic deletion and its key management mechanism as well as present the Micro Content Management system as an example application for cryptographic deletion. In the demo scenario, we will create new storage containers with and without cryptographic deletion and show data ingestion and retrieval using different client applications. Our Bluebox user interface features interactive visualizations of the key management data structures. We will use the visualization to show how insertions/deletions affect the key management data structure. We will demonstrate the proposed separation of encrypted data and unencrypted metadata by extracting metadata from previously ingested emails, pictures and documents. We then show how the tasks for metadata extraction and replication are triggered, how the metadata warehouse tables look like, and how analyses can be realized.

Screenshots of the application, as well as more details about its capabilities, can be found on our github page: `https://github.com/timwaizenegger/mcm-bluebox`

References

[Ba16] Barney, Jonathan; Lebutsch, David; Mega, Cataldo; Schleipen, Stefan; Waizenegger, Tim: Deletion of content in digital storage systems. US Patent 9,298,951, March 2016.

[DW10] Diesburg, Sarah M.; Wang, An-I Andy: A Survey of Confidential Data Storage and Deletion Methods. ACM Comput. Surv., 43(1):2:1–2:37, December 2010.

[EP16] European Parliament, Council: Regulation (EU) 2016/679: General Data Protection Regulation. Article 17 "Right to Erasure", 2016.

[Os15] Osborne, Charlie: Ashley Madison hack: How much user data did "Paid delete" function obliterate? ZDNet Security, 2015.

[Wa17] Waizenegger, Tim; Lebutsch, David; Mega, Cataldo; Mitschang, Bernhard: Design and Implementation of Efficient Key Management for Secure Cryptographic Data Erasure in Large Cloud-Based Storage Systems. Under review for: EDBT 20th International Conference on Extending Database Technology, 2017.

B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 629

The Big Picture: Understanding large-scale graphs using Graph Grouping with GRADOOP

Martin Junghanns[1] André Petermann[1] Niklas Teichmann[2] Erhard Rahm[1]

Abstract: Graph grouping supports data analysts in decision making based on the characteristics of large-scale, heterogeneous networks containing millions or even billions of vertices and edges. We demonstrate graph grouping with GRADOOP, a scalable system supporting declarative programs composed from multiple graph operations. Using social network data, we highlight the analytical capabilities enabled by graph grouping in combination with other graph operators. The resulting graphs are visualized and visitors are invited to either modify existing or write new analytical programs. GRADOOP is implemented on top of Apache Flink, a state-of-the-art distributed dataflow framework, and thus allows us to scale graph analytical programs across multiple machines. In the demonstration, programs can either be executed locally or remotely on our research cluster.

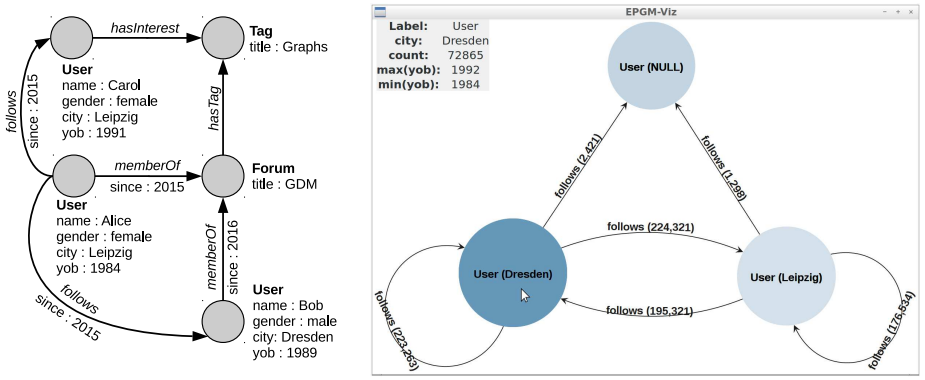
Keywords: Graph Analytics, Graph Algorithms, Distributed Computing, Dataflow systems

1 Introduction

Graphs are an intuitive way to model, analyze and visualize complex relationships among real-world data objects. The flexibility of graph data models and the variety of existing graph algorithms made graph analytics attractive to different domains, e.g., to analyze the link structure of the world wide web [BP98], users of a social network [Ne10], protein interaction in biological networks [Pa11] or business process executions in enterprise data [Pe14]. In these domains, graphs are often heterogeneous in terms of the objects they represent. For example, vertices of a social network may represent users and forums while edges may express friendships or memberships. Further on, vertices and edges may have associated properties to describe the respective object, e.g., a user's name or the date a user became member of a forum.

The property graph model [RN10, An12] is an established approach to model heterogeneous networks. Figure 1(a) shows a property graph that represents a simple social network containing multiple types of vertices (e.g., *User* and *Forum*) and edges (e.g., *follows* and *memberOf*). Vertices as well as edges are further described by properties in the form of key-value pairs (e.g., *name : Alice* or *since : 2015*). However, while small graphs are an intuitive way to visualize connected information, with vertex and edge numbers increasing up to millions or billions, it becomes almost impossible to understand the encoded information by mere visual inspection. One way to reduce complexity is the grouping of vertices and edges to so-called *super vertices* and *super edges* of a *summary graph* supporting the analyst in extracting and understanding the underlying information [THP08, Ch08, Ju17].

[1] University of Leipzig, Database Group & ScaDS Dresden/Leipzig,
[junghanns,petermann,rahm]@informatik.uni-leipzig.de
[2] University of Leipzig, Database Group & ScaDS Dresden/Leipzig, teichmann@studserv.uni-leipzig.de



(a) Example of a Social Network

(b) Summary graph in EPGM-Viz; Properties are shown for the selected vertex

Fig. 1: (a) shows an example social network; (b) shows the summary graph of a large social network grouped by the user's city including aggregate values expressing the oldest and youngest user's age per city and the number of edges among cities.

Graph grouping allows structural summarization and attribute aggregation by user-defined vertex and edge properties. In Figure 1(b), the users of a social network are grouped by their *city* property, i.e., each super vertex in the summary graph represents all users that live in the same city. As the property graph model is schemaless, users may not necessarily provide certain properties. Such users are grouped within a dedicated *NULL* vertex. Super edges represent mutual relationships among the grouped vertices of the input graph. For example, an edge between Dresden and Leipzig in the summary graph represents all edges of that same type among users from Dresden and Leipzig. Besides the structural summarization, graph grouping allows for attribute aggregation on super vertices and super edges. In our example, each super vertex and super edge stores the number of elements it represents. Super vertices additionally store the minimum and maximum year of birth among the users.

In the demonstration, we present the graph grouping operator [Ju17] of Gradoop [Ju16], an open-source graph analytical system that implements the so-called Extended Property Graph Model and that supports declarative operations on single property graphs as well as collections of these. For example, given a social network similar to Figure 1(a), the summary graph of Figure 1(b) can be easily declared by the following script:

```
summaryGraph = socialNetwork
  .subgraph(
    (vertex -> vertex[:label] == 'User'),
    (edge -> edge[:label] == 'follows'))
  .groupBy(
    [:label, 'city'], [COUNT(), MIN('yob'), MAX('yob')],
    [:label], [COUNT()])
```

We will demonstrate how Gradoop and its operators can be used to compute expressive summary graphs. Since Gradoop is implemented on top of Apache Flink [Ca15], a distributed state-of-the-art dataflow framework, demo programs can be executed either locally or on our research cluster without modifying the program. In the following section, we provide a brief overview about Gradoop and the Extended Property Graph Model. In Section 3, we describe our demonstration scenario in more detail.

2 Graph Grouping with GRADOOP

GRADOOP is a system for declarative graph analytics supporting the combination of multiple graph operators and algorithms in a single program. Graph data is represented within the Extended Property Graph Model (EPGM) [Ju16], which is based on the property graph model [RN10], i.e., on directed multigraphs supporting identifiers, labels and named attributes (properties) for vertices as well as edges. As an extension, the EPGM supports the concept of *logical graphs*, which are logical partitions of a base graph. Thus, it is possible to analyze single graphs as well as collections of these. Logical graphs also support labels and properties, for example, to represent communities in a social network and to store their number of users. Furthermore, logical graphs and collections are input and output of EPGM operators which enables the composition of complex analytical programs. GRADOOP already provides operator implementations for graph pattern matching, subgraph extraction, graph transformation, set operations on multiple graphs as well as property-based aggregation and selection [Ju16]. Graph analytical programs are declared using a Java API representing the domain specific language GrALa (Graph Analytical Language). Below the user-facing API, graph operators and algorithms are mapped to the programming abstractions provided by Apache Flink and thus, their execution can be scaled out across a cluster of machines. The source code of GRADOOP is available online under GPL license[4].

Graph grouping extends the set of available graph operators in GRADOOP. The operator takes a single logical graph as input and computes a new logical graph, which we call a *summary graph*. The operator signature for graph grouping in GrALa is defined as follows:

```
LogicalGraph.groupBy(
  vertexGroupingKey[], vertexAggregateFunction[],
  edgeGroupingKey[], edgeAggregateFunction[]) : LogicalGraph
```

While the first argument is a list of vertex grouping keys, the second argument refers to a list of user-defined or system-provided vertex aggregate functions. Analogously, the third and fourth argument are used to define edge grouping keys and edge aggregate functions. In our introductory example, the operator is parameterized using the symbol `:label` and the property key `city` as vertex grouping keys. We use system-provided aggregate functions to count the elements inside each group and to determine minimum and maximum year of birth. The resulting super vertices adopt the label, the grouping property (e.g., *city : Dresden*) and the results of the aggregate functions (e.g., *count : 72,865*). Edges are implicitly grouped by the super vertices of their incident vertices and explicitly by their label and counted. In the example, one can also see the composition of subgraph extraction and graph grouping. First, a subgraph containing solely vertices of type *User* and edges of type *follows* is extracted from the social network and forwarded to the grouping operator. The resulting summary graph can be either used as input for another operator (e.g., pattern matching), stored in a data sink or visualized.

[4] `http://www.gradoop.com`

3 Demonstration Description

In our demonstration, we show how Gradoop can be used to compute summary graphs from social network data. We provide three example programs to cover distinct aspects of graph grouping: (1) subgraph grouping analogous to our example, (2) type-dependent grouping to declare grouping keys on individual labels and (3) graph grouping along dimensional hierarchies for Graph OLAP scenarios [Ch08]. Example (2) and (3) require the graph transformation operator to pre-process the input graph before the actual grouping. Visitors are also invited to modify our example programs or to write new ones including further operators (e.g., pattern matching) and plug-in algorithms (e.g., community detection).

The programs are presented and developed using the Gradoop Java API and executed either locally on the demonstration laptop or remotely on our research cluster. We provide real-world graphs and artificial social network data with up to 10 billion edges generated by the LDBC data generator [Er15]. Gradoop provides multiple ways to visualize the resulting summary graphs. In example (1), we will use our graph visualization EPGM-Viz[5] (Figure 1(b)), in example (2), we will use the DOT output and GraphViz[6], and in example (3), we will utilize the Neo4j graph database to visualize the results[7].

4 Acknowledgments

This work is partially funded by the German Federal Ministry of Education and Research under project ScaDS Dresden/Leipzig (BMBF 01IS14014B).

References

[An12] Angles, R.: A Comparison of Current Graph Database Models. In: Proc. ICDEW. 2012.

[BP98] Brin, Sergey; Page, Lawrence: The Anatomy of a Large-scale Hypertextual Web Search Engine. In: Proc. WWW. 1998.

[Ca15] Carbone, P. et al.: Apache Flink™: Stream and Batch Processing in a Single Engine. IEEE Data Eng. Bull., 38(4), 2015.

[Ch08] Chen, C.; Yan, X.; Zhu, F.; Han, J.; Yu, P. S.: Graph OLAP: Towards online analytical processing on graphs. In: Proc. ICDM. 2008.

[Er15] Erling, O. et al.: The LDBC Social Network Benchmark. In: Proc. SIGMOD. 2015.

[Ju16] Junghanns, M.; Petermann, A.; Teichmann N.; Gómez K.; Rahm E.: Analyzing Extended Property Graphs with Apache Flink. In: Proc. SIGMOD NDA Workshop. 2016.

[Ju17] Junghanns, M.; Petermann, A.; Rahm E.: Distributed Graph Grouping with Gradoop. In: Proc. BTW. 2017.

[Ne10] Newman, M.: Networks: An Introduction. 2010.

[Pa11] Pavlopoulos, G. A. et al.: Using graph theory to analyze biological networks. BioData Mining, 4(1), 2011.

[Pe14] Petermann, A.; Junghanns, M.; Müller, R.; Rahm, E.: BIIIG: Enabling business intelligence with integrated instance graphs. In: Proc. ICDE Workshops. 2014.

[RN10] Rodriguez, M. A.; Neubauer, P.: Constructions from Dots and Lines. arXiv, 2010.

[THP08] Tian, Y.; Hankins, R. A.; Patel, J. M.: Efficient Aggregation for Graph Summarization. In: Proc. SIGMOD. 2008.

[5] https://github.com/dbs-leipzig/EPGM-Viz

[6] http://www.graphviz.org/

[7] https://github.com/s1ck/flink-neo4j

Autorenverzeichnis