

GESELLSCHAFT
FÜR INFORMATIK



Kai-Uwe Sattler, Melanie Herschel,
Wolfgang Lehner (Hrsg.)

**Datenbanksysteme für
Business, Technologie und Web
(BTW 2021)**

**13.–17. September 2021
in Dresden, Deutschland**

Gesellschaft für Informatik e. V. (GI)

Lecture Notes in Informatics (LNI) — Proceedings

Series of the Gesellschaft für Informatik (GI)

Volume P-311

ISBN 978-3-88579-705-0

ISSN 1617-5468

Volume Editors

Kai-Uwe Sattler

Technische Universität Ilmenau
FG Datenbanken und Informationssysteme
98684 Ilmenau, Deutschland
Email: kus@tu-ilmenau.de

Melanie Herschel

Universität Stuttgart
Institut für Parallele und Verteilte Systeme
70569 Stuttgart, Deutschland
Email: melanie.herschel@ipvs.uni-stuttgart.de

Wolfgang Lehner

Technische Universität Dresden
Professur für Datenbanken
01062 Dresden, Deutschland
Email: wolfgang.lehner@tu-dresden.de

Series Editorial Board

Andreas Oberweis, Karlsruher Institut für Technologie (KIT), Deutschland
(Chairman, oberweis@kit.edu)

Dieter Fellner, Technische Universität Darmstadt, Deutschland

Ulrich Flegel, Infineon, Deutschland

Ulrich Frank, Universität Duisburg-Essen, Deutschland

Michael Goedicke, Universität Duisburg-Essen, Deutschland

Ralf Hofestädt, Universität Bielefeld, Deutschland

Wolfgang Karl, KIT Karlsruhe, Deutschland

Michael Koch, Universität der Bundeswehr München, Deutschland

Peter Sanders, Karlsruher Institut für Technologie (KIT), Deutschland

Andreas Thor, HfT Leipzig, Deutschland

Ingo Timm, Universität Trier, Deutschland

Karin Vosseberg, Hochschule Bremerhaven, Deutschland

Maria Wimmer, Universität Koblenz-Landau, Deutschland

Dissertations

Steffen Hölldobler, Technische Universität Dresden, Deutschland

Thematics

Agnes Koschmider, Universität Kiel, Deutschland

Seminars

Andreas Oberweis, Karlsruher Institut für Technologie (KIT), Deutschland

© Gesellschaft für Informatik, Bonn 2021

printed by Köllen Druck+Verlag GmbH, Bonn



This book is licensed under a

Creative Commons Attribution-NonCommercial 3.0 License.

Vorwort

Die “BTW-Konferenz” stellt innerhalb der deutschsprachigen Daten-Management-Community eines der zentralen Events zum wissenschaftlichen Austausch dar. Seit über 30 Jahren wird die Serie der Fachtagungen “Datenbanksysteme für Business, Technologie und Web” (BTW) des Fachbereichs “Datenbanken und Informationssysteme” (DBIS) der Gesellschaft für Informatik (GI) im zweijährigen Rhythmus an unterschiedlichen Orten ausgetragen. Für 2021 war die Veranstaltung vom 08. - 12. März an der Technischen Universität Dresden unter der Schirmherrschaft des Ministerpräsidenten des Freistaates Sachsen, Michael Kretschmer, geplant – Konferenzräume waren gebucht, Keynote-Rednerin und –Redner eingeladen, umfangreiche Sponsorenunterstützung sichergestellt und – auch wichtig für eine BTW – ein attraktives Begleitprogramm ausgestaltet. Leider hat die Corona-Pandemie die Durchführung einer Präsenzveranstaltung zu dem ursprünglich geplanten Zeitpunkt unmöglich gemacht.

Ungeachtet dessen fand jedoch ein Begutachtungsprozess der eingereichten wissenschaftlichen Arbeiten statt, dessen Ergebnis in dem nun vorliegenden Band dokumentiert ist. Erstmals wurden dabei umfangreiche Änderungen zur Sicherung der Qualität (im weiteren Sinne) eingeführt. Die sichtbarste Neuerung besteht in der Einführung eines Track-Systems und damit einhergehend der Integration des Industrie-Tracks in den regulären Begutachtungsprozess. Durch das thematisch organisierte Track-System konnte nun sichergestellt werden, dass die Rückmeldungen der Gutachterinnen und Gutachter zielgenau auf die Positionierung eines Beitrags passten. Folgende Tracks wurden definiert:

- Database Technology (Alexander Böhm; SAP SE)
- ML & Data Science (Matthias Böhm; TU Graz)
- Data Integration, Semantic Data Management, Streaming (Katja Hose; Aalborg University)
- (Industrial) Use Cases & Applications (Stefanie Scherzinger; Universität Passau)

Der gesamte Begutachtungsprozess wurde von den beiden PC-Chairs Melanie Herschel (Universität Stuttgart) und Kai-Uwe Sattler (Technische Universität Ilmenau) organisiert. Als Proceedings Chair hat Alexander Krause (TU Dresden) zum Gelingen des vorliegenden Bandes beigetragen. Als weitere Neuerung wurde der Review-Prozess so gestaltet, dass Revisionen von Einreichungen ermöglicht wurden, so dass Anmerkungen und Hinweise der Gutachterinnen und Gutachter in die wissenschaftlichen Arbeiten noch mit aufgenommen und entsprechend berücksichtigt werden konnten. Dieser explizite und bewusst sehr stark gelebte Revisionsprozess hat zu einer deutlichen Verbesserung des gesamten Begutachtungsablaufs geführt. Als letzte – durch einen Stempel an den jeweiligen Beiträgen markiert - sichtbare Neuerung darf an dieser Stelle noch auf den Reproduzierbarkeitsprozess hingewiesen werden. Alle angenommenen Arbeiten wurden eingeladen, die in

den Publikationen dokumentierten experimentellen Ergebnisse durch ein Mitglied der Community wiederholen zu lassen. Diese als Mentoring und bewusst nicht als Kontrolle positionierte Wiederholung wurde für knapp ein Drittel der in diesem Band publizierten Beiträge durchgeführt – im internationalen Vergleich ein Riesenerfolg und Zeugnis einer lebhaften und offenen Data-Management-Community!

So wichtig wie die Publikation eines wissenschaftlichen Ergebnisses ist, so notwendig ist auch die Präsentation und eine sich anschließende offene Diskussion. Zum aktuellen Zeitpunkt kann leider auf Grund der sehr volatilen Infektionslage noch keine klare Aussage getroffen werden, ob wir – wie aktuell geplant – vom 13. bis 17. September 2021 eine BTW an der Technischen Universität Dresden durchführen können. Auf jeden Fall wird es eine Präsentation der angenommenen Beiträge geben, wenn in Präsenz nicht durchführbar dann notwendigerweise in einem virtuellen Rahmen. In diesem Kontext ist somit auch auf die Website der BTW2021 hinzuweisen (<https://btw2021-dresden.de/>), auf welcher neben aktuellen Informationen auch alle weiteren Materialien zur Verfügung stehen.

Unabhängig von den weiteren Entwicklungen darf an dieser Stelle bereits den vielen Menschen gedankt sein, die an der Vorbereitung der BTW2021 beteiligt waren und sind; im Einzelnen geht unser Dank an

- alle Kolleginnen und Kollegen, die sich als Gutachterinnen und Gutachter aktiv in den Begutachtungsprozess eingebracht haben oder sich in der Organisation von Studierendenprogramm, Workshop- und Tutorienprogramm, Demoprogramm, Data-Science Challenge und Dissertationspreis engagiert haben.
- alle Partner und Sponsoren, die uns trotz Pandemie eine umfangreiche Unterstützung zugesagt haben.
- die GI-Geschäftsstelle für die Unterstützung bei der Finanzplanung und –abwicklung.
- die TU Dresden, die Fakultät Informatik der TU Dresden und den Lehrstuhl für Datenbanken für tatkräftige Unterstützung in der organisatorischen Vorbereitung – angefangen von der Öffentlichkeitsarbeit bis hin zur - hoffentlich benötigten – Bereitstellung der Räumlichkeiten.
- und - last but not at least – an alle Autoren und Autorinnen, ohne deren Input in Form wissenschaftlicher Beiträge eine BTW2021 nicht denkbar wäre!

Vielen Dank bereits jetzt an alle Beteiligten!

Dresden, im Februar 2021

Wolfgang Lehner (TU Dresden), Tagungsleitung

Johann-Christoph Freytag (HU Berlin), Ehrenvorsitzender

Ulrike Schöbel, Dirk Habich, Maik Thiele (TU Dresden), Lokale Organisation und Finanzen

Tagungsleitung

Wolfgang Lehner, Technische Universität Dresden

Organisationskomitee

Ulrike Schöbel, Technische Universität Dresden

Dirk Habich, Technische Universität Dresden

Maik Thiele, Technische Universität Dresden

Ehrervorsitzender

Johann-Christoph Freytag, Humboldt-Universität zu Berlin

Wissenschaftliches Programmkomitee

Vorsitzende

Melanie Herschel, Universität Stuttgart

Kai-Uwe Sattler, Technische Universität Ilmenau

Track Chairs

Alexander Böhm, SAP SE

Matthias Böhm, Technische Universität Graz

Katja Hose, Aalborg University

Stefanie Scherzinger, Universität Passau

Mitglieder

Ziawasch Abedjan, Leibniz Universität Hannover
Carsten Binnig, Technische Universität Darmstadt
Stefan Conrad, Heinrich-Heine-Universität Düsseldorf
Stefan Deßloch, Technische Universität Kaiserslautern
Jens Dittrich, Universität des Saarlandes
Florian Funke, Snowflake Inc, San Francisco
Michael Gertz, Universität Heidelberg
Anika Groß, Hochschule Anhalt
Michael Grossniklaus, Universität Konstanz
Torsten Grust, Universität Tübingen
Alfons Kemper, Technische Universität München
Meike Klettke, Universität Rostock
Birgitta König-Ries, Friedrich-Schiller-Universität Jena
Wolfgang Lehner, Technische Universität Dresden
Ulf Leser, Humboldt-Universität zu Berlin
Stefan Mandl, Exasol AG Nürnberg
Stefan Manegold, Centrum Wiskunde & Informatica (CWI), Amsterdam
Norman May, SAP SE, Walldorf
Sebastian Michel, Technische Universität Kaiserslautern
Thomas Neumann, Technische Universität München
Daniela Nicklas, Universität Bamberg
Marcus Paradies, DLR, Jena
Tilmann Rabl, HPI, Potsdam
Erhard Rahm, Universität Leipzig
Norbert Ritter, Universität Hamburg
Gunter Saake, Otto-von-Guericke-Universität Magdeburg
Kai-Uwe Sattler, Technische Universität Ilmenau
Harald Schöning, Software AG, Darmstadt
Holger Schwarz, Universität Stuttgart
Bernhard Seeger, Philipps-Universität Marburg
Thomas Seidl, Ludwig-Maximilians-Universität München
Christin Seifert, Universität Duisburg-Essen
Günther Specht, Universität Innsbruck
Knut Stolze, IBM Research & Development, Böblingen
Uta Störl, Hochschule Darmstadt
Jens Teubner, Technische Universität Dortmund
Jonas Traub, Technische Universität Berlin
Gottfried Vossen, Universität Münster
Lena Wiese, Fraunhofer ITEM, Hannover
Wolfram Wingerath, Baqend, Hamburg

Inhaltsverzeichnis

Wissenschaftliche Beiträge

Database Technology

Alexander Kumaigorodski, Clemens Lutz, Volker Markl <i>Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing</i>	19
Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, Alfons Kemper <i>B²-Tree: Cache-Friendly String Indexing within B-Trees</i>	39
Julian Weise, Sebastian Schmidl, Thorsten Papenbrock <i>Optimized Theta-Join Processing</i>	59
Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, Michael Grossniklaus <i>Precise, Compact, and Fast Data Access Counters for Automated Physical Database Design</i>	79
Jonas Dann, Daniel Ritter, Holger Fröning <i>Exploring Memory Access Patterns for Graph Processing Accelerators</i>	101
Lukas Karnowski, Maximilian E. Schüle, Alfons Kemper, Thomas Neumann <i>Umbra as a Time Machine: Adding a Versioning Type to SQL</i>	123

ML & Data Science

Lucas Woltmann, Claudio Hartmann, Dirk Habich, Wolfgang Lehner <i>Aggregate-based Training Phase for ML-based Cardinality Estimation</i>	135
--	-----

Nico Lässig, Sarah Oppold, Melanie Herschel <i>Using FALCES against bias in automated decisions by integrating fairness in dynamic model ensembles</i>	155
Sandra Obermeier, Anna Beer, Florian Wahl, Thomas Seidl <i>Cluster Flow — an Advanced Concept for Ensemble-Enabling, Interactive Clustering</i>	175
Steffen Kläbe, Stefan Hagedorn <i>Applying Machine Learning Models to Scalable DataFrames with Grizzly</i>	195
 Data Integration, Semantic Data Management, Streaming	
Stefan Lerm, Alieh Saeedi, Erhard Rahm <i>Extended Affinity Propagation Clustering for Multi-source Entity Resolution</i>	217
Benjamin Warnke, Sven Groppe, Muhammad Waqas Rehan, Stefan Fischer <i>Flexible data partitioning schemes for parallel merge joins in semantic web queries</i>	237
Ziad Sehili, Florens Rohde, Martin Franke, Erhard Rahm <i>Multi-Party Privacy Preserving Record Linkage in Dynamic Metric Space</i>	257
Elena Beatriz Ouro Paz, Eleni Tzirita Zacharatou, Volker Markl <i>Towards Resilient Data Management for the Internet of Moving Things . .</i>	279
Kevin Gomez, Matthias Täschner, M. Ali Rostami, Christopher Rost, Erhard Rahm <i>Graph Sampling with Distributed In-Memory Dataflow Systems</i>	303
Aslihan Özmen, Mahdi Esmailoghli, Ziawasch Abedjan <i>Combining Programming-by-Example with Transformation Discovery from large Databases</i>	313
Sven Langenecker, Christoph Sturm, Christian Schalles, Carsten Binnig <i>Towards Learned Metadata Extraction for Data Lakes</i>	325

Tanja Auge, Andreas Heuer <i>Tracing the History of the Baltic Sea Oxygen Level</i>	337
---	-----

(Industrial) Use Cases & Applications

Corinna Giebler, Christoph Gröger, Eva Hoos, Rebecca Eichler, Holger Schwarz, Bernhard Mitschang <i>The Data Lake Architecture Framework</i>	351
--	-----

Lars Gleim, Liam Tirpitz, Stefan Decker <i>FactStack: Interoperable Data Management and Preservation for the Web and Industry 4.0</i>	371
---	-----

Wolfgang Mauerer, Ralf Ramsauer, Edson R. Lucas, D. Lohmann, Stefanie Scherzinger <i>Silentium! Run–Analyse–Eradicate the Noise out of the DB/OS Stack . . .</i>	397
--	-----

Daniel Glake, Fabian Panse, Norbert Ritter, Thomas Clemen, Ulfia Lenfers <i>Data Management in Multi-Agent Simulation Systems</i>	423
---	-----

Liste der Autorinnen und Autoren

Wissenschaftliche Beiträge

Database Technology

Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing

Alexander Kumaigorodski¹, Clemens Lutz², Volker Markl³



Abstract: Comma-separated values (CSV) is a widely-used format for data exchange. Due to the format's prevalence, virtually all industrial-strength database systems and stream processing frameworks support importing CSV input.

However, loading CSV input close to the speed of I/O hardware is challenging. Modern I/O devices such as InfiniBand NICs and NVMe SSDs are capable of sustaining high transfer rates of 100 Gbit/s and higher. At the same time, CSV parsing performance is limited by the complex control flows that its semi-structured and text-based layout incurs.

In this paper, we propose to speed-up loading CSV input using GPUs. We devise a new parsing approach that streamlines the control flow while correctly handling context-sensitive CSV features such as quotes. By offloading I/O and parsing to the GPU, our approach enables databases to load CSVs at high throughput from main memory with NVLink 2.0, as well as directly from the network with RDMA. In our evaluation, we show that GPUs parse real-world datasets at up to 76 GB/s, thereby saturating high-bandwidth I/O devices.

Keywords: CSV; Parsing; GPU; CUDA; RDMA; InfiniBand

1 Introduction

Sharing data requires the data provider and data user to agree on a common file format. *Comma-separated values (CSV)* is currently the most widely-used format for sharing tabular data [DMB17, Ne17, Me16]. Although alternative formats such as Apache Parquet [Ap17] and Albis [Te18] exist, in the future the CSV format will likely remain popular due to continued advocacy by open data portals [KH15, Eu20]. As a result, database customers request support for loading terabytes of CSV data [Oz18]. Fast data loading is necessary to reduce the delay before the data are ready for analysis.

Fresh data are typically sourced either from disk or streamed in via the network, thus *loading* the data consists of device I/O and parsing the file format [Me13]. However, recent advances in I/O technologies have lead to a *data loading bottleneck*. RDMA network interfaces and NVMe storage arrays can transfer data at 12.5 GB/s and beyond [Be16, Ze19]. Research suggests that CPU-based parsers cannot ingest CSV data at these rates [Ge19, SJ20].

¹ TU Berlin, Germany, alxkum@gmail.com

² DFKI GmbH, Berlin, Germany, clemens.lutz@dfki.de

³ TU Berlin & DFKI GmbH, Germany, volker.markl@tu-berlin.de

In this paper, we investigate how I/O-connected GPUs enable fast CSV loading. We stream data directly to the GPU from either main memory or the network using fast GPU interconnects and GPUDirect. Fast GPU interconnects, such as AMD Infinity Fabric [AM19], Intel CXL [CX19], and Nvidia NVLink [Nv17], provide GPUs with high bandwidth access to main memory [Le20b], and GPUDirect provides GPUs with direct access to RDMA and NVMe I/O devices [Nv20a]. Furthermore, next-generation GPUs will be tightly integrated into RDMA network cards to form a new class of *data processing unit (DPU)* devices [Nv20b]. To parse data at high bandwidth, we propose a new GPU- and DPU-optimized parsing approach. The key insight of our approach is that multiple data passes in GPU memory simplify complex control flows, and increase computational efficiency.

In summary, our contributions are as follows:

- (1) We propose a new approach for fast, parallel CSV parsing on GPUs (Section 3).
- (2) We provide a new, streamed loading strategy that uses GPUDirect RDMA [Nv20a] to transfer data directly from the network onto the GPU (Section 4).
- (3) We evaluate the impact of a fast GPU interconnect for end-to-end streamed loading from main memory and back again (Section 5). We use NVLink 2.0 to represent the class of fast GPU interconnects.

The remainder of this paper is structured as follows. In Section 2, we give a brief overview of performing I/O on GPUs, and of related work on CSV parsing. Then, we describe our contributions to CSV parsing and streaming I/O in Sections 3 and 4. Next, we evaluate our work in Section 5, and discuss our findings on loading data using GPUs in Section 6. Finally, we give our concluding remarks in Section 7.

2 Background and Related Work

In this section, we describe how GPUDirect RDMA and fast GPU interconnects enable high-speed I/O on GPUs. We then give an overview of CSV parsing, and differentiate our approach from related work on CSV loading.

2.1 I/O on GPUs

GPUs are massively parallel processors that run thousands of threads at a time. The threads are executed by up to 80 streaming multiprocessors (*SMs*) on the Nvidia “Volta” architecture [Nv17]. Each SM runs threads in *warps* of 32 threads, that execute the same instruction on multiple data items. Branches that cause control flow to diverge thus slow down execution (*warp divergence*). Within a warp, threads can exchange data in registers using *warp shuffle* instructions. Up to 32 warps are grouped as a *thread block*, that can exchange data in *shared memory*. The GPU also has up to 32 GB of on-board *GPU memory*.

I/O on the GPU is typically conducted via a PCIe 3.0 interconnect that connects the GPU to the system at 16 GB/s. Recently, fast interconnects have emerged that provide system-wide cache-coherence and, in the case of NVLink 2.0, up to 75 GB/s per GPU [IB18]. Databases usually use these interconnects to transfer data between main memory and GPU memory, thus linking the GPU to the CPU. However, *GPUDirect RDMA* and *GPUDirect Storage* connect the GPU directly to an *I/O device*, such as an RDMA network interface (e.g., InfiniBand) or an NVMe storage device (e.g., a flash disk). This connection bypasses the CPU and main memory by giving the I/O device direct memory access to the GPU’s memory. Although the data bypasses the CPU, the CPU orchestrates transfers and GPU execution.

GPUDirect Storage has been used in a GPU-enabled database to manage data on flash disks [Le16]. In contrast, we propose to load data from external sources into the database.

In summary, fast interconnects and GPUDirect enable the GPU to efficiently perform I/O. In principle, these technologies can be combined. However, due to our hardware setup, in our experimental evaluation we distinguish between NVLink 2.0 to main memory, and GPUDirect RDMA with InfiniBand via PCIe 3.0.

2.2 CSV Parsing

CSV is a tabular format. The data are logically structured as records and fields. Thus, parsers split the CSV data at record or field boundaries to facilitate later deserialization of each field. The parser determines the structure by parsing field (‘,’) and record (‘\n’) delimiters, as CSV files provide no metadata mapping from its logical structure to physical bytes. Quotes (‘”’) make parsing more difficult, as delimiters within quotes are literal characters and do not represent boundaries. The CSV format is formalized in RFC4180 [Sh05], although variations exist [DMB17].

Parsing CSV input in parallel involves splitting the data into *chunks* that can be parsed by independent threads. Initially, the parser splits the file at arbitrary bytes, and then adjusts the split offsets to the next delimiter [Me13]. Detecting the correct quotation context requires a separate data pass, most easily performed by a single thread [Me13]. As quotes come in pairs, parallel *context detection* first counts all quotes in each chunk, and then performs a prefix sum to determine if a quote opens (odd) or closes (even) a quotation [Ea16, Ge19].

CPU parsers have been optimized by eliminating data passes through speculative context detection [Ge19, Le17], and replacing complex control flows with SIMD data flows [Ge19, LL19, Le17, Me13]. In contrast to these works, we optimize parsing for the GPU by applying data-parallel primitives (i.e., prefix sum) and by transposing the data into a columnar format. These optimizations reduce warp divergence on GPUs, but add two data passes for a total of 3 passes (without context detection) or 4 passes (with context detection). We reduce the overhead of these additional passes by caching intermediate data in GPU memory.

Stehle and Jacobsen have presented a GPU-enabled CSV parser [SJ20]. Their parser tracks multiple finite state machines to enable a generalization to other data formats, e.g., JSON or XML. Our evaluation shows that this generality is computation-intensive and limits throughput. In contrast, we explore loading data directly from an I/O device and a fast GPU interconnect. These technologies require a fast CSV parsing approach. We thus minimize computation by specializing our approach to RFC4180-compliant CSV data. However, our approach is capable of handling CSV dialects [DMB17] by allowing users to specify custom delimiters and quotation characters at runtime. In addition, our approach can detect certain errors with no performance penalty, e.g., CSV syntax issues involving “ragged” rows with missing fields and cell-level issues such as numerical fields containing units. Detected issues can be logged and reported to the user.

3 Approach

In this section we introduce a new algorithm, *CUDAFastCSV*, for parsing CSV data that is optimized for GPUs. Optimizing for GPUs is challenging, because parsers typically have complex control flows. However, fast GPU kernels should regularize control flow to avoid execution penalties caused by warp divergence. Therefore, our approach explores a new trade-off: we simplify control flow at the expense of additional data passes, and exploit the GPU’s high memory bandwidth to cache the data during these passes. This insight forms the basis on which we adapt CSV parsing to the GPU architecture.

We first give a conceptual overview of our approach in Figure 1. Next, we describe and discuss each step in more detail with its challenges and solutions in the following subsections.

Conceptually, the CSV input is first transferred to GPU memory from a data source, e.g., main memory or an I/O device. The input is then split into equally sized chunks to be processed in parallel. With the goal to index all field positions in the input data, we first count the delimiters in each chunk and then create prefix sums of these counted delimiters. Using the prefix sums, the chunks are processed again to create the *FieldsIndex*. This index allows the input data to be copied to column-based *tapes* in the next step. Tapes enable

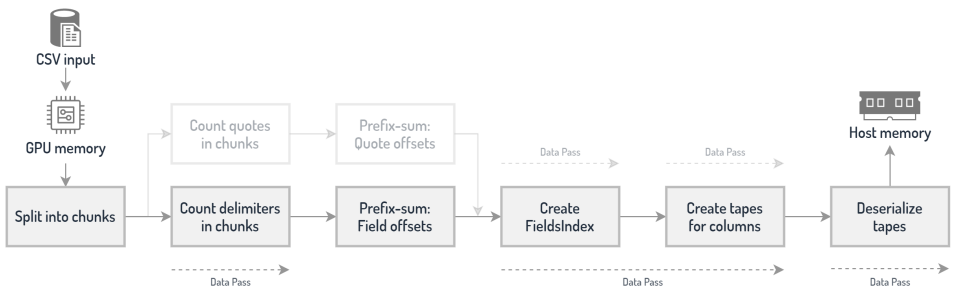


Fig. 1: Conceptual overview of our approach

us to vectorize processing by transposing multiple rows into a columnar format. Creating the FieldsIndex and tapes are logically separate steps, but can be fused into a single data pass. Finally, each tape is deserialized in parallel. The resulting data are column-oriented and can then be further processed on the GPU or copied to another destination for further processing, e. g., to the host’s main memory.

In this default *Fast Mode*, the parser is unaware of the context and correct quotation scope when fields are enclosed in quotation marks. To create a context-aware FieldsIndex, we introduce the *Quoted Mode*, as fields may themselves contain field delimiters. Quoted Mode is an alternative parsing mode that additionally keeps track of quotation marks. Quoted Mode allows us to parallelize parsing of quoted CSV data, but it is more processing intensive than the default parsing mode. However, as well-known public data sources indicate that quotes are rarely used in practice⁴⁵, the main focus of our work is on the Fast Mode.

In this section, we assume the CSV input already resides in GPU memory. In Section 4, we present *Streaming I/O*, which allows incoming chunks of data to be parsed without the need for the entire input data to be in GPU memory.

Overall, our data-parallel CSV parser solves three main challenges: (1) splitting the data into chunks for parallel processing, (2) determining each chunk’s context, and (3) vectorized deserialization of fields with their correct row and column numbers.

3.1 Parallelization Strategy

Parsing data on GPUs requires massive parallelism to achieve high throughput. In the following, we explain how we parallelize CSV parsing in our approach.

Simply parallelizing by rows requires iterating over all data first. It also results in unevenly sized row lengths. This causes subsequent parsing or deserialization threads in a warp to stall during individual processing, thus limiting hardware utilization. Instead, Figure 2 shows how we split the input data at fixed offsets to get equally sized chunks. These chunks

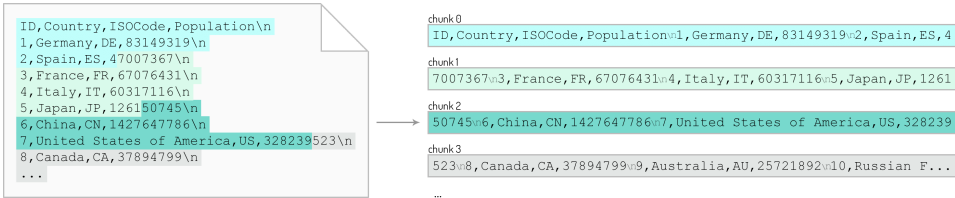


Fig. 2: Splitting input into equally sized, independent, chunks

⁴Kaggle. <https://www.kaggle.com/datasets?filetype=csv>

⁵NYC OpenData. <https://data.cityofnewyork.us/browse?limitTo=datasets>

are independent of each other and individually processed by a warp. This avoids threads from becoming idle as they transfer and process the same amount of data.

The choice of *chunk size* and how a warp loads and reads its chunks impacts the parallelizability of the delimiter-counting process and, ultimately, the entire parsing process. Because coalesced byte-wise access is not enough to saturate the available bandwidth, we read four bytes per thread, which correspond to the 128 bytes of a GPU memory transaction per warp. However, we experimentally find that looping over multiple consecutive 128-byte chunks per warp increases bandwidth even more, compared to increasing the number of thread blocks. This reduces the pressure on the GPU’s warp scheduler and the CUDA runtime.

To identify chunk sizes that allow for optimal loading and processing, we evaluate several kernels that each load chunks of different sizes. We discover that casting four consecutive bytes to an `int` and loading the `int` into a register is more efficient than loading the bytes one at a time. For input that is not a multiple of 128 bytes, a challenge here is to efficiently avoid a memory access violation in the last chunk. Using a branch condition for bounds-checking takes several cycles to evaluate. We avoid a branch altogether by padding the input data with `NULL`s to a multiple of 128 bytes during input preparation. Conventionally, strings are `NULL`-terminated, thus any such occurrence simply causes these padded bytes to be ignored during loading and later parsing.

3.2 Indexing Fields

We can now start processing the chunks. Our goal in this phase is to index all of the field positions of the input data in the *FieldsIndex*. This index is an integer array of yet unknown size `rows*columns` with a sequence of continuous field positions. We construct the *FieldsIndex* in three steps.

In the first pass over the chunks, every warp counts the field delimiters in its chunk. The number of delimiters in each chunk is stored in an array. For optimization purposes, record delimiters are treated as field delimiters, thus, creating a continuous sequence of fields.

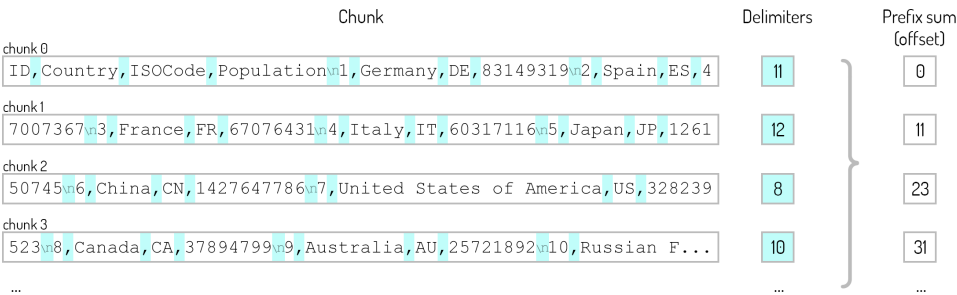


Fig. 3: Computing the field offset for every chunk using a prefix sum

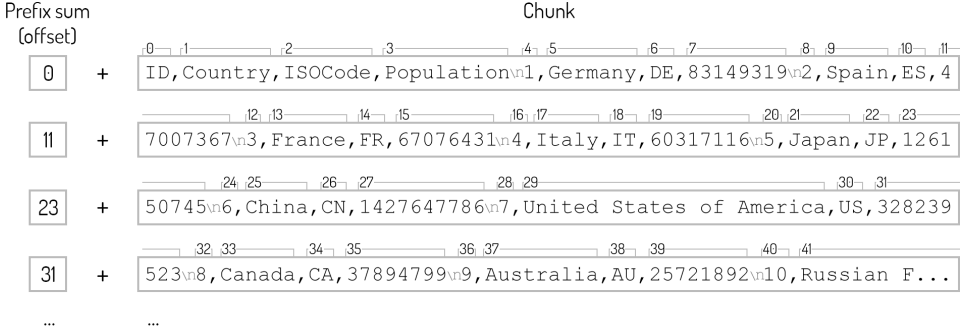


Fig. 4: Using the chunk's prefix sum to infer field positions

In the second phase, we compute the chunks' field offsets with an exclusive prefix sum, as illustrated in Figure 3. At the end of the prefix sum calculation, the total number of fields is automatically available. We divide the number of fields by the number of columns to obtain the number of rows, and allocate the necessary space for the FieldsIndex array in GPU memory. The number of columns is specified in the table schema.

In the third and final phase, the FieldsIndex can now be filled in parallel. We perform a second pass over all the chunks and scan for field delimiters again. As shown in Figure 4, the total number of preceding fields in the input data can instantly be inferred using the prefix sum at a chunk's position.

For a thread to correctly determine a field's index when encountering its delimiter, however, it also needs to know the total number of delimiters in the warp's preceding threads. Thus, for every 128 byte loop iteration over the chunk, threads first count how many delimiters they have in their respective four byte sector. Since threads within a warp can efficiently access each other's registers, calculating an exclusive prefix sum of these numbers is fast. These prefix sums provide the complete information needed to determine a field's exact position and index to store it in the FieldsIndex array. The length of a field can also be inferred from the FieldsIndex.

3.2.1 Quoted Mode

For the *Quoted Mode*, additional steps are required to create a correct FieldsIndex.

When counting delimiters in the first phase, quotation marks are simultaneously counted in a similar manner. After calculating the prefix sums for the delimiters, the prefix sums for the quotation marks are created as well. In the third phase, during the second pass over the chunks when counting delimiters again, quotation marks are also counted again, and prefix sums are created for both within the warp.

```
ID,Name,Philosophy\n1,"Aristotle","Quality is not an act, it is a habit."\n2,"Plato","When men speak ill of thee, live so as nobody may believe them."\n3,"Epictetus","It's not what happens to you, but how you react to it that matters."\n...
```

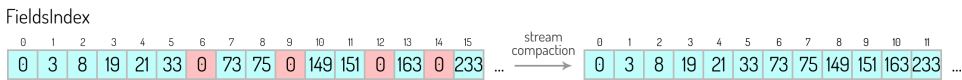


Fig. 5: Additional pass in Quoted Mode to remove invalid delimiters

We then exploit the fact that a character is considered quoted whenever the number of preceding quotation marks is uneven. Before writing a field's position into the FieldsIndex when encountering a record or field delimiter, first the number of total preceding quotation marks at this position is checked. Should that number be uneven, a sentinel value of 0 is written to the FieldsIndex at the index that the field's position would otherwise have been written to. The sentinel value represents an invalid delimiter. This approach also allows for quotation symbols inside fields since, in accordance with RFC4180, quotation marks that are part of the field need to be escaped with another quotation mark.

After the FieldsIndex is created, a stream compaction pass is done on the FieldsIndex to remove all invalid, i. e., quoted, delimiters and remove gaps between valid, i. e., unquoted, delimiters. We illustrate an example with valid and invalid field delimiters in Figure 5. Separating this additional step from the actual FieldsIndex creation simplifies control flow and helps to coalesce writes to memory.

3.3 Deserialization

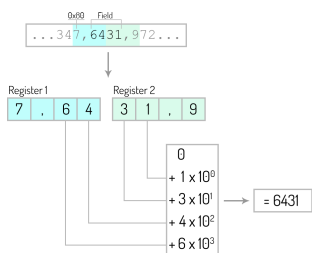


Fig. 6: Deserialization

Efficient deserialization on the GPU is a many-sided problem. Not only is the question of how to vectorize deserialization challenging, but also how to keep the entire warp occupied. A simple approach is to have every thread deserialize a field. However, we must assume that neighboring columns have different data types. Constructing a generic kernel that can handle all data types involves many branches, causing warp divergence. Additionally, using any row-based approach requires adding lots of complexity to work in parallel. Complexity that is likely to cause idle threads. Any

approach that is column-based, however, can make use of the fact that all fields in the column have the same data type, thus, giving us an easy pattern to vectorize. Every thread in the warp deserializes one field, allowing the entire warp to deserialize 32 fields in parallel. Similar to SQL's DDL (Data Definition Language), users specify a column's maximum

length along its type for deserialization purposes. To keep the warp’s memory access pattern coalesced, every thread first consecutively reads four aligned bytes into a dedicated register until enough bytes were read to satisfy the specified length of the column. If all column fields are contiguous in memory, the warp is likely able to coalesce memory accesses. In a loop equal to the size of the specified column length, every thread can now read and convert each digit from a register while calculating the running sum, as illustrated in Figure 6.

While this approach can lead to workload imbalances within a warp, i. e., when neighboring fields in a warp have unequal lengths, this approach causes no warp divergence and only uses one branch in the entire kernel.

3.4 Optimizing Deserialization: Transposing to Tapes

Since our deserializer uses a column-based approach, its memory access pattern only allows for a coalesced and aligned memory access with full use of all the relevant bytes when given the optimal circumstances. CSV, however, is a row-oriented storage format. The optimal circumstances would only come into effect when there is just one column or the field’s data types are identical along a multiple of 32 wide field count. To improve deserialization performance for columns with various data types we introduce deserialization with tapes. *Tapes* are buffers in which the parser temporarily stores fields in a column-oriented layout. The column-oriented layout enables vectorized deserialization of fields.

We illustrate our approach with an example in Figure 7. A separate tape for every column is created in an additional step during the parsing process. We assume that the length of each column is specified by the table’s schema (e.g., `CHAR(10)`). We then define a tape’s width (*tapeWidth*) equal to its specified column length. For every field in the FieldsIndex, the input’s field value is copied to its column’s tape at an offset equal to the field’s row number:

$$tapeAddress(field) := tape_{col(field)} + row(field) \times tapeWidth_{col(field)}$$

Field values that do not fully use their *tapeWidth* are right-padded with NULLs on the tape.

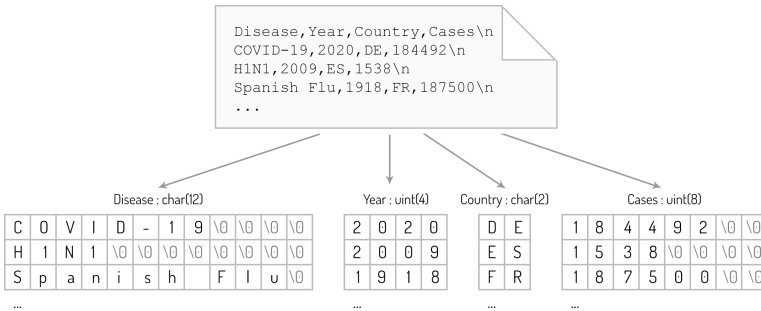


Fig. 7: Visual representation of deserialization tapes

For our Fast Mode, materializing the entire `FieldsIndex` in GPU memory can be skipped and instead the chunk's `FieldsIndex` is temporarily written to shared memory. When a chunk's `FieldsIndex` is complete, the field values can be directly copied onto the tapes. However, the length of the chunk's last field cannot be calculated from the chunk's `FieldsIndex` alone. Instead, we work around this obstacle by saving each chunk's first delimiter offset along the chunk's delimiter count during the first step of the parsing process. Combining these two steps saves us from materializing the `FieldsIndex` and from having to do a total of four passes over the input data. However, we cannot perform this optimization in the Quoted Mode, as the complete `FieldsIndex` is required to detect the quotation context.

4 Streaming I/O

We extend our approach to allow *streaming* of partitioned input data. This enables us to start parsing the input before it is fully copied onto the GPU, i.e., reducing overall latency, and for input that is too big to otherwise fit into the GPU's memory.

The input is split into *partitions* before being copied to the GPU's memory for individual and independent parsing without the need for the complete input data to be on the GPU. The partitions are equal in length and of size *streamingPartitionSize*.

4.1 Context Handover

In typesetting, *widows* are lines at the end of a paragraph left dangling at the top from the previous page. *Orphans* are lines at the start of a paragraph left dangling at the bottom for the next page. Both are separated from the rest of their paragraph. Partitioning our input data creates a similar effect that we need to account for, as illustrated in Figure 8. In a partition, we consider the last row an orphan, which will not be parsed. Instead we copy the orphan's bytes to a widow buffer. The next partition prepends available data from the widow buffer to its partition data before starting the parsing process. The widow buffer's size is defined by a configuration variable. We set its default size to 10 KB, which is sufficient to handle single rows spanning over 10,000 characters.

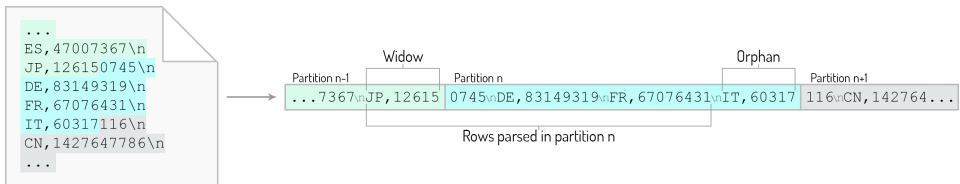


Fig. 8: Widows are taken from the previous partition, while orphans are left for the next partition

4.2 End-to-End Loading

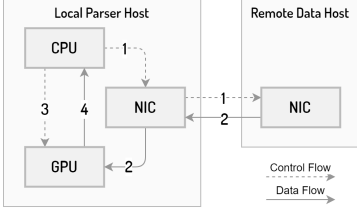


Fig. 9: A WorkStream’s control and data flow for its partition

We realize streaming as *WorkStream* items in our implementation, representing a CUDA stream and a partition for processing. Every WorkStream item has a dedicated *partitionBuffer* in the GPU’s device memory that is used to copy its partition’s chunks into. For RDMA input data, this *partitionBuffer* is also automatically registered for GPUDirect transfers via RDMA.

In Figure 9, we show the four states of a WorkStream. (1) First, an available WorkStream requests a remote memory read, which (2) the remote host responds to by copying the requested data directly into GPU memory. Afterwards, (3) the local CPU schedules a kernel for the WorkStream for parsing. Finally, (4) the kernel is synchronized after finishing and the partition’s result data can be optionally transferred to main memory.

5 Evaluation

In this section we evaluate the performance of parsing CSV data on GPUs.

5.1 Experiment Setup

In the following, we give an overview of our experimental evaluation environment.

Hardware. We use two identical machines for the majority of our testing (Intel Xeon Gold 5115, 94 GB DDR4-2400, Nvidia Tesla V100-PCIe with 16 GB HBM2, Mellanox MT27700 InfiniBand EDR, Ubuntu 16.04). A third machine was used for NVLink related evaluations (IBM AC922 8335-GTH, 256 GB DDR4-2666, Nvidia Tesla V100-SXM2 with 16 GB HBM2, Ubuntu 18.04). For all our tests, we use only one NUMA node, i.e., a single GPU and CPU with their respective memory.

Methodology. We measure the mean and standard error over ten runs with the help of high-resolution timers. For GPU-related measurements, we adhere to Nvidia’s recommendations when benchmarking CUDA applications [FJ19]. The time for the initialization of processes, CUDA, or memory, is not included in these measurements. All input files are read from the Linux in-memory file system *tmpfs*. With the exception of NVLink-related measurements, we note that our measurements are stable with a standard error of less than 5% from the mean. We measure the throughput in GB/s.

Datasets. For our evaluations we use a real-world dataset (*NYC Yellow Taxi Trips Jan-Mar 2019*, 1.9 GB, 22.5M records, 14 numerical fields out of 18, short and consistent record lengths), a standardized dataset (*TPC-H Lineitem 2.18.0*, 719 MB, 6M records,

16 fields of various data types and string fields with varying lengths), and a synthetic dataset (*int_444*, 1 GB, 70M records, three fields of four random digits).

Databases and Parsers. In Section 5.2.2, we compare CUDAFastCSV in Fast Mode to CPU and GPU baselines. *OmniSciDB* (v5.1.2), *PostgreSQL* (v12.2), *HyPer DB* (v0.5), *ParPaRaw* [SJ20], *RAPIDS cuDF* (v0.14.0), and *csvmonkey* (v0.1). *PostgreSQL* and *csvmonkey* are single-threaded, all other baselines parse in parallel on all CPU cores or on the GPU. Except for *PostgreSQL* and *ParPaRaw*, we explicitly disable quotation parsing.

I/O. In Section 5.2.3, we stream the input data from four I/O sources to compare performance against the potentially transfer bound parsing from Section 5.2.2. We stream data over interconnects and InfiniBand using two datasets. In contrast to end-to-end parsing, results are not copied back to the host’s main memory. *On-GPU* serves as a baseline with the input data already residing in GPU memory. *PCIe 3.0* serves as an upper bound for I/O devices on the host. *NVLink 2.0* is, in comparison to *PCIe 3.0*, a higher-bandwidth and lower-latency alternative [Le20b]. In *RDMA with GPUDirect* one machine acts as the file server, while another machine with CUDAFastCSV in Fast Mode streams the input data using RDMA directly into the GPU’s memory using GPUDirect, bypassing the CPU and main memory.

5.2 Results

In this section, we present our performance results and comment on our observations.

5.2.1 Tuning Parameters

In this section, we evaluate the parameters for performance tuning and scalability that we introduce in Section 3.

Chunk Size. The choice of the *chunkSize* in CUDAFastCSV determines how much of the input data a warp processes. An increasing size requires more hardware resources per warp but also reduces the overhead associated with scheduling, launching, and processing new thread blocks or warps. Figure 10 shows the throughput as a function of the chunk size. The

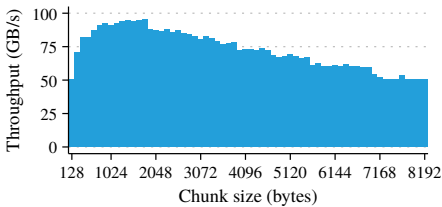


Fig. 10: Impact of *chunkSize* on *int_444*

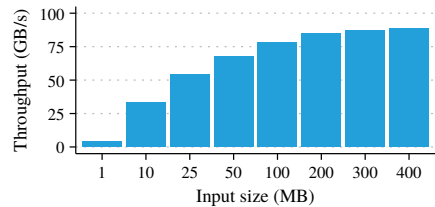


Fig. 11: Performance of input size for *int_444*

observed performance reflects our design discussion in Section 3.1. The memory bandwidth for small chunks initially increases when processing 128-byte multiples, but then drops, e. g., before 2048 bytes. The drop stems from one less concurrent thread block running on the SM due to a lack of available shared memory resources. A slight rise in performance before every drop shows the improved resource utilization of the available resources. We conclude that the best chunk size is 1024 bytes.

Input Size. Figure 11 shows the ramp up of CUDAFastCSV’s performance when given an increasingly larger input file. While the 1 MB file only achieves 4.5 GB/s, the throughput already strongly increases with a 10 MB file to 33.6 GB/s and continues to rise until it approaches its limit of approximately 90 GB/s. We conclude that performance scales quickly with regard to the input size, and maximum throughput is approached at 100 MB.

Streaming Size. In Figure 12, we define a baseline of approximately 12 GB/s for PCIe 3.0 as it represents the maximum possible throughput for that machine. In our results, the throughput scales almost linearly with the *streamingPartitionSize* up until 10 MB before it hits its maximum of 11 GB/s at 20 MB. For comparison, we include results from the same experiment over NVLink 2.0. Ramp-up speed is very similar to PCIe 3.0 but keeps rising when the limitations of PCIe 3.0 would otherwise set in. In contrast, with NVLink 2.0 we achieve a peak throughput of 48.3 GB/s. Thus, NVLink 2.0 is 4.4× faster than PCIe 3.0. However, our implementation is not able to achieve NVLink’s peak bandwidth due to the limited amount of DMA copy engines, and due to the overhead from data and buffer management required for streaming. This leads to delays, as transfers and compute are not fully overlapped. We conclude that PCIe 3.0’s bandwidth is saturated quickly and its best *streamingPartitionSize* is already achieved at 20 MB. NVLink 2.0 exposes PCIe 3.0 as a bottleneck for end-to-end parsing in comparison.

Warp Index Buffer Size. The *warpIndexBufferSize* parameter in CUDAFastCSV limits the maximum number of found fields in all chunk segments within a warp and is used to reserve the kernel’s shared memory space in Fast Mode or, in Quoted Mode, the required space in global memory for the FieldsIndex. It can be altered from its default, 2048 bytes, to increase parallelism when the underlying data characteristics of the CSV input data allow for it. As such, less shared memory resources are allocated per thread block, allowing for additional

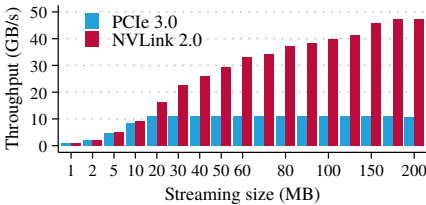


Fig. 12: Impact of parameter *streamingPartitionSize* on int_444

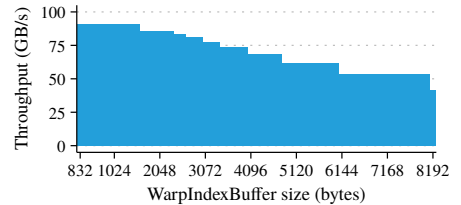


Fig. 13: Impact of oversized *warpIndexBufferSize* on int_444

thread blocks to run concurrently on the SM. Figure 13 illustrates this behavior as the amount of concurrent thread blocks steps down whenever the increasing size allocates too many resources. For a chunk size of 1024, the smallest viable *warpIndexBufferSize* for the *int_444* dataset is 832. Maximum throughput of around 90.9 GB/s is kept up until 1536. The default of 2048 falls into the 85.6 GB/s range. To accommodate for a worst-case scenario of only having empty fields in a 1024 byte chunk, we would need a *warpIndexBufferSize* of 4096, which reduces our performance to 68.6 GB/s. Larger sizes reduce performance even further. We conclude that the *warpIndexBufferSize* has a large impact on performance, as it is dependent on the underlying structure of the input data.

5.2.2 Databases and Parsers

To evaluate end-to-end parsing performance of CUDAFastCSV, we benchmarked our approach in Fast Mode against several implementations from different categories as described in our experiment setup. We use the *NYC Yellow Taxi* and *TPC-H* dataset, residing in the host’s main memory, and measure the time until all deserialized fields are available in the host’s main memory in either a row- or a column-oriented data storage format.

NYC Yellow Taxi. The performance numbers reported for parsing and deserializing the 1.9 GB dataset in Figure 14 highlight the strength of CUDAFastCSV, which is only limited by the PCIe 3.0’s available bandwidth. This is especially noteworthy, as deserialization includes nine floating point numbers and five integers out of the 18 total fields. The GPU-based implementation, *cuDF* with its new and updated CSV implementation, achieves only a quarter of the performance of CUDAFastCSV. Our approach is at least 4x times faster than all CPU-based approaches, i. e., *PostgreSQL*, *HyPer DB*, *OmniSciDB*, *csvmonkey*, and **Instant Loading* (measured by Stehle and Jacobsen [SJ20] using 32 CPU cores). CUDAFastCSV over NVLink 2.0 more than triples the performance compared to PCIe 3.0.

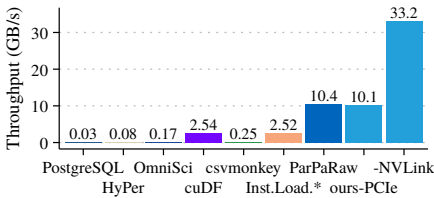


Fig. 14: *Taxi* end-to-end performance

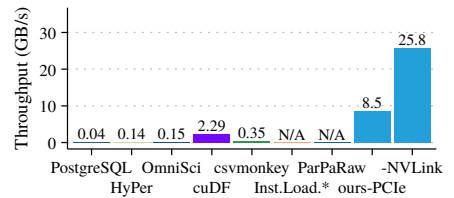


Fig. 15: *TPC-H* end-to-end performance

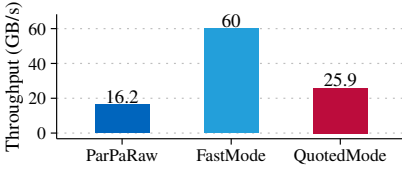


Fig. 16: On-GPU throughput of *ParPaRaw* and *CUDAFastCSV*

Only *ParPaRaw* provides comparable performance to *CUDAFastCSV*. To determine if *ParPaRaw* is being limited by the interconnect in this instance, we additionally measured its on-GPU throughput for this dataset and compared it to our implementation in Figure 16. In comparison to *ParPaRaw*, our Quoted Mode is 1.6x faster and our Fast Mode is even 3.7x faster. The reason is that we are able to reduce

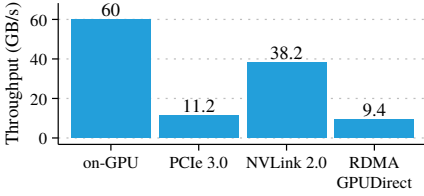
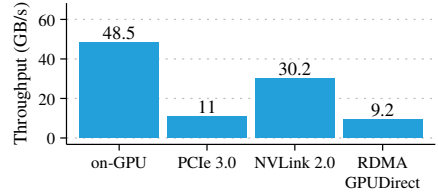
the overall amount of work, as we do not need to track multiple state machines, and our approach is less processing-intensive as a result.

TPC-H Lineitem. Figure 15 shows *CUDAFastCSV* to have a slightly lower throughput when compared to the previous dataset on both PCIe 3.0 and NVLink 2.0. The bottleneck for this data set is the transfer of the larger result data back to the host, causing increasingly longer delays between streamed partitions. For every 100 MB partition of *TPC-H* data transferred to the GPU, approximately 118 MB of result data need to be transferred back to the host, while the *NYC Yellow Taxi* data only need 93 MB per 100 MB. This causes delays in input streaming and during processing, as kernel invocations are hindered by data dependencies and synchronization. *cuDF*, another GPU-based implementation, shows a similar drop in performance of approximately 10%. In contrast, some of the CPU-based implementations were able to significantly improve their performance for the *TPC-H* dataset, namely *HyPer DB* and *csvmonkey*, due to the smaller number of numeric fields that need to be deserialized. NVLink 2.0 again more than triples the performance of *CUDAFastCSV* in comparison to PCIe 3.0.

5.2.3 I/O

We present results for *CUDAFastCSV* when streamed over two interconnects and InfiniBand. We evaluate performance using the *NYC Yellow Taxi* and *TPC-H Lineitem* datasets to compare against the potentially transfer bound end-to-end parsing.

NYC Yellow Taxi. The baseline for Figure 17 is 60 GB/s, representing *CUDAFastCSV*’s maximum possible performance over an interconnect to the GPU. As seen in the previous section, while our implementation over PCIe 3.0 can fully saturate the bus, it is still less than a fifth of the on-GPU performance. Again, throughput over NVLink 2.0 more than triples and shows the limitations of the PCIe 3.0 system in comparison. Our RDMA with GPUDirect approach, streaming the input data from a remote machine directly onto GPU memory over the internal PCIe 3.0 bus, is at an expected sixth of the on-GPU performance. It is unclear why the GPUDirect connection is slower than the local copy, as the network is not the bottleneck. Li et al. [Le20a] present similar results, and suggest that PCIe P2P access might be limited by the chipset.

Fig. 17: *Taxi* I/O streaming performanceFig. 18: *TPC-H* I/O streaming performance

TPC-H Lineitem. The baseline is established in Figure 18 with 48.5 GB/s. Similarly to the taxi dataset, PCIe 3.0 is saturated but only at a sixth of the on-GPU performance, while NVLink 2.0 performance is almost triple in comparison. For the RDMA with GPUDirect approach we achieve similar performance for the *TPC-H* dataset. Overall, throughput for this dataset is slightly lower for the baseline and for every interconnect, due to the increased size of the result data and its consequences as described in the previous section.

5.2.4 Quoted Mode

The Quoted Mode is an alternative parsing mode that keeps track of quotation marks to create a context-aware FieldsIndex. In contrast to the Fast Mode, the Quoted Mode involves additional processing steps. We show a comparison between the two modes for three datasets in Figure 19. For all three datasets, the Quoted Mode has roughly half the throughput of the Fast Mode. The cause of this performance drop is the materialization of the FieldsIndex in GPU memory combined with a subsequent stream compaction pass, which are avoided in Fast Mode. We observe that the performance drops more for the *NYC Yellow Taxi* dataset than for the *TPC-H* and *int_444* datasets. The reason is that fields have less content on average in *NYC Yellow Taxi*, thus the FieldsIndex is larger in proportion to the data size (i.e., more delimiters per MB of data). Nevertheless, throughput is still higher than that of other implementations in our comparisons.

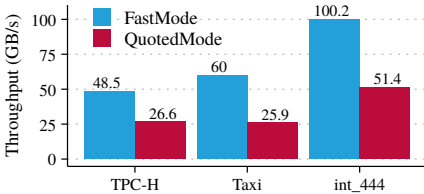


Fig. 19: Fast Mode vs. Quoted Mode

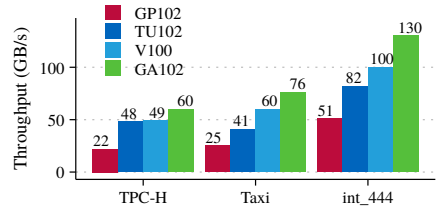


Fig. 20: Comparison across generations

5.2.5 Hardware Scalability

In Figure 20, we compare four Nvidia GPU generations to assess the performance impact of hardware evolution: Pascal, Turing, Volta, and Ampere. The lineup includes the server-grade Tesla V100-PCIe GPU, and three high-end desktop-grade GPUs (Nvidia GTX 1080 Ti with 11 GB GDDR5X, RTX 2080 Ti with 11 GB GDDR6, and RTX 3080 with 10 GB GDDR6X). We measure the parsing throughput of our three datasets, with the data stored in GPU memory. We observe that the throughput incrementally speeds up by factors of 1.61–2.18, 1.02–1.46, and 1.22–1.3 between the respective generations. The total increase from Pascal to Ampere is 2.55–3.02 times.

To explain the reasons for the speed-up, we profile the parser on the Tesla V100. Profiling shows that building the FieldsIndex and transposing to tapes accounts for 85% of the execution time. The main limiting factor of this kernel are execution stalls caused by instruction and memory latency. For the TPC-H dataset, warp divergence causes additional overhead. Thus, throughput increases mainly due to the higher core counts (more in-flight instructions) and clock speeds (reduced instruction latency) of newer GPUs. In contrast, higher bandwidth at identical compute power (Volta vs. Turing, both having 14 TIPS for Int32) only yields a significant speed-up when there is little warp divergence.

6 Discussion

In this section, we discuss the lessons we learned from our evaluation.

GPUs improve parsing performance. In comparison to a strong CPU baseline, our measurements show that parsing on the GPU still improves throughput by 13x for the *NYC Yellow Taxi* dataset. Compared to a weak baseline, throughput can even be improved by 73x for the *TPC-H Lineitem* dataset. Thus, offloading parsing to the GPU can provide significant value for databases.

Fast parsing of quoted data. We show that our approach is able to parallelize context detection in Quoted Mode, and scale performance up to 51 GB/s. At this throughput, we are near the peak bandwidth of NVLink 2.0.

Interconnect bandwidth limits performance. In all our measurements, PCIe 3.0 does not provide sufficient bandwidth to achieve peak throughput. Using NVLink 2.0 instead, the throughput increases by 2.8–3.4x. This improvement shifts the bottleneck to our pipelining strategy. Removing this limitation would increase throughput further by 1.6x.

Network streaming is feasible. We show that streaming data from the network to the GPU is possible and provides comparable performance to loading data from the host’s main memory over PCIe 3.0. This strategy provides an interesting building block for data streaming frameworks.

GPUs can efficiently handle complex data format features. Features, such as quoted fields, decrease parsing throughput to 43-55% of the non-quoted throughput. However, this reduced throughput is still higher than the bandwidth provided by PCIe 3.0 and InfiniBand. Thus, the overall impact is no loss in performance. Only for faster I/O devices, e.g., 400 Gbit/s InfiniBand, would Quoted Mode become a bottleneck.

GPUs facilitate data transformation. We show that GPUs efficiently transform row-oriented CSV data into the column-oriented layout required by in-memory databases. As saturating the I/O bandwidth requires only a fraction of the available compute resources, GPUs are well-positioned to perform additional transformations for databases [No20].

Desktop-grade GPUs provide good performance per cost. For all our datasets, a desktop-grade GPU is sufficient to saturate the PCIe 3.0 interconnect. At the same time, desktop-grade GPUs cost only a fraction of server-grade GPUs (7000 EUR for a Tesla V100 compared to 1260 EUR MSRP for a RTX 2080 Ti in 2021). Thus, buying a server-grade GPU only makes sense for extra features such as NVLink 2.0 and RDMA with GPUDirect.

7 Conclusion

In this work, we explore the feasibility of loading CSV data close to the transfer rates of modern I/O devices. Current InfiniBand NICs transfer data at up to 100 Gbit/s, and multiple devices can be combined to scale the bandwidth even higher. Our analysis shows that CPU-based parsers cannot process data fast enough to saturate such I/O devices, which leads to a data loading bottleneck.

To achieve the required parsing throughput, we leverage GPUs by using a new parsing approach and by connecting the GPU directly to the I/O device. Our implementation demonstrates that GPUs reach a parsing throughput of up to 100 GB/s for data stored in GPU memory. In our evaluation, we show that this is sufficient to saturate current InfiniBand NICs. Furthermore, our NVLink 2.0 measurements underline that GPUs are capable of scaling up to emerging 200 and 400 Gbit/s I/O devices. We envision that in the future, loading data directly onto the GPU will free up computational resources on the CPU, and will thus enable new opportunities to speed-up query processing in databases and stream processing frameworks.

In conclusion, I/O-connected GPUs are able to solve the data loading bottleneck, and represent a new way with which database architects can integrate GPUs into databases.

Acknowledgments

We thank Elias Stehle for sharing and helping us to measure ParPaRaw. This work was funded by the EU Horizon 2020 programme as E2Data (780245), the DFG priority programme “Scalable Data Management for Future Hardware” (MA4662-5), the German Ministry for Education and Research as BBDC (01IS14013A) and BIFOLD — “Berlin Institute for

the Foundations of Learning and Data” (01IS18025A and 01IS18037A), and the German Federal Ministry for Economic Affairs and Energy as Project ExDra (01MD19002B).

Bibliography

- [AM19] AMD: AMD EPYC CPUs, AMD Radeon Instinct GPUs and ROCm Open Source Software to Power World’s Fastest Supercomputer at Oak Ridge National Laboratory. <https://www.amd.com/en/press-releases/2019-05-07-amd-epyc-cpus-radeon-instinct-gpus-and-rocm-open-source-software-to-power>, Accessed: 2019-07-05, May 2019.
- [Ap17] Apache Software Foundation: Apache Parquet. <https://parquet.apache.org/>, Accessed: 2020-10-08, October 2017.
- [Be16] Binnig, Carsten; et al.: The End of Slow Networks: It’s Time for a Redesign. PVLDB, 2016.
- [CX19] CXL: Compute Express Link Specification Revision 1.1. <https://www.computeexpresslink.org>, June 2019.
- [DMB17] Döhmen, Till; Mühleisen, Hannes; Boncz, Peter A.: Multi-Hypothesis CSV Parsing. In: SSDBM. 2017.
- [Ea16] Eads, Damian: ParaText: A library for reading text files over multiple cores. <https://github.com/wiseio/paratext>, Accessed: 2020-09-09, 2016.
- [Eu20] European Commission & Open Data Institute: European Data Portal e-Learning Programme. <https://www.europeandataportal.eu/elearning/en/module9/#/id/co-01>, Accessed: 2020-08-31, March 2020.
- [FJ19] Fiser, Bill; Jodłowski, Sebastian: Best Practices When Benchmarking CUDA Applications. In: GTC - GPU Tech Conference. 2019.
- [Ge19] Ge, Chang; et al.: Speculative Distributed CSV Data Parsing for Big Data Analytics. PVLDB, 2019.
- [IB18] IBM POWER9 NPU team: Functionality and performance of NVLink with IBM POWER9 processors. IBM Journal of Research and Development, 62(4/5):9, 2018.
- [KH15] Kim, James G; Hausenblas, Michael: 5-Star Open Data. <https://5stardata.info>, Accessed: 2020-08-31, 2015.
- [Le16] Li, Jing; et al.: HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. PVLDB, 2016.
- [Le17] Li, Yanan; et al.: Mison: A Fast JSON Parser for Data Analytics. PVLDB, 2017.
- [Le20a] Li, Ang; et al.: Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. IEEE Trans. Parallel Distrib. Syst., 2020.
- [Le20b] Lutz, Clemens; et al.: Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. SIGMOD, 2020.
- [LL19] Langdale, Geoff; Lemire, Daniel: Parsing gigabytes of JSON per second. VLDB J., 2019.

- [Me13] Mühlbauer, Tobias; et al.: Instant Loading for Main Memory Databases. PVLDB, 2013.
- [Me16] Mitlöhner, Johann; et al.: Characteristics of Open Data CSV Files. OBD, 2016.
- [Ne17] Neumaier, Sebastian; et al.: Data Integration for Open Data on the Web. In: Reasoning Web. Lecture Notes in Computer Science, 2017.
- [No20] Noll, Stefan; Teubner, Jens; May, Norman; Boehm, Alexander: Shared Load(ing): Efficient Bulk Loading into Optimized Storage. In: CIDR. 2020.
- [Nv17] Nvidia: Nvidia Tesla V100 GPU Architecture (Whitepaper). <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>, Accessed: 2019-03-26, 2017.
- [Nv20a] Nvidia: GPUDirect. <https://developer.nvidia.com/gpudirect>, Accessed: 2020-09-08, 2020.
- [Nv20b] Nvidia: Nvidia Introduces New Family of BlueField DPUs. <https://nvidianews.nvidia.com/news/nvidia-introduces-new-family-of-bluefield-dpus-to-bring-breakthrough-networking-storage-and-security-performance-to-every-data-center>, Accessed: 2021-01-18, 2020.
- [Oz18] Ozer, Stuart: How to Load Terabytes into Snowflake — Speeds, Feeds and Techniques. <https://www.snowflake.com/blog/how-to-load-terabytes-into-snowflake-speeds-feeds-and-techniques>, Accessed: 2020-08-31, April 2018.
- [Sh05] Shafranovich, Yakov: RFC 4180. <https://tools.ietf.org/pdf/rfc4180.pdf>, Accessed: 2019-03-29, 2005.
- [SJ20] Stehle, Elias; Jacobsen, Hans-Arno: ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. PVLDB, 2020.
- [Te18] Trivedi, Animesh; et al.: Albis: High-Performance File Format for Big Data Systems. In (Gunawi, Haryadi S.; Reed, Benjamin, eds): USENIX ATC. 2018.
- [Ze19] Zeuch, Steffen; et al.: Analyzing Efficient Stream Processing on Modern Hardware. PVLDB, 2019.

B²-Tree: Cache-Friendly String Indexing within B-Trees.

Josef Schmeißer,¹ Maximilian E. Schüle,² Viktor Leis,³ Thomas Neumann,⁴ Alfons Kemper⁵

Abstract: Recently proposed index structures, that combine trie-based and comparison-based search mechanisms, considerably improve retrieval throughput for in-memory database systems. However, most of these index structures allocate small memory chunks when required. This stands in contrast to block-based index structures, that are necessary for disk-accesses of beyond main-memory database systems such as Umbra. We therefore present the B²-tree. The outer structure is identical to that of an ordinary B+-tree. It still stores elements in a dense array in sorted order, enabling efficient range scan operations. However, B²-tree is composed of multiple trees, each page integrates another trie-based search tree, which is used to determine a small memory region where a sought entry may be found. An embedded tree thereby consists of decision nodes, which operate on a single byte at a time, and span nodes, which are used to store common prefixes. This architecture usually accesses fewer cache lines than a vanilla B+-tree as shown in our performance evaluation. As a result, the B²-tree answers point queries considerably faster.

Keywords: Indexing; B-tree; String

1 Introduction

Low overhead buffer managers are a fairly recent development which provide in-memory performance in case the data does fit into RAM [Le18; NF20]. However, database systems based on such a low overhead buffer manager still require efficient index structures which harness this new architecture. While systems like HyPer [KN11] could use pure in-memory based index structures, like the Adaptive Radix Tree (ART) [LKN13] or the more recent Height Optimized Trie (HOT) [B18], these are no longer an option for Umbra [NF20]. Pure in-memory index structures usually offer better performance than various B-tree flavors, yet their tendency to allocate small varying sized memory chunks limits their range of applicability.

With the presentation of LeanStore [Le18], Leis et al. revisited the role of buffer managers. LeanStore is a storage engine designed to resolve the overhead issues of traditional buffer management architectures [Ha08]. Its main feature is to abandon a hash table based pinning

¹ TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching josef.schmeisser@tum.de

² TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching m.schuele@tum.de

³ Friedrich Schiller University Jena, Ernst-Abbe-Platz 2, 07743 Jena viktor.leis@uni-jena.de

⁴ TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching neumann@in.tum.de

⁵ TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching kemper@in.tum.de

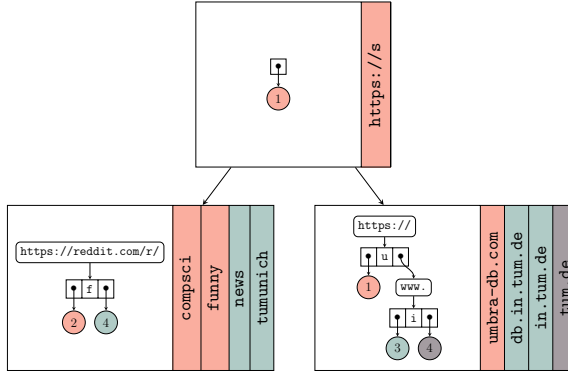


Fig. 1: The B^2 -tree consists of decision nodes, similar to B+-tree nodes that contain separators and pointers to sub-nodes, and span nodes for common prefixes.

architecture, which buffer managers usually use, in favor of a technique called *pointer swizzling* [GUW09]. Umbra’s buffer manager extends this concept by the ability to serve variable-sized pages with a minimum page size of 64 KiB [NF20]. This obviously affects the architectural requirements imposed on index structures. The imposed constraint precludes the use of most state-of-art pure in-memory based index structures. B-trees and their variations, on the other hand, fit well into Umbra’s architecture. However, we found that even a highly optimized B+-tree implementation is no longer competitive, with regard to string indexing, in comparison to index structures like ART and HOT. Our B^2 -tree operates on top of Umbra’s buffer manager and provides significant throughput improvements over the original Umbra B+-tree.

Fig. 1 shows a small B^2 -tree. It hosts an embedded tree per B-tree page. This embedded tree serves the purpose of directing incoming searches into narrowed down search spaces. A search on the embedded tree yields a pair of slot indices which define a span wherein a sought key may be found. Circular nodes point to the beginning of a search range, the upper bound. Each search space is also highlighted by the distinct coloring of its records and the corresponding node within the embedded tree.

Modern CPU architectures usually provide three layers of cache between their registers and main memory in order to mitigate the imbalance between CPU performance and main memory latency [SPB05]. Performing a naïve binary search over all the entries stored on a reasonably large B-tree page usually results in high lookup costs. This is especially the case when the B-tree page is used to store variable-sized records. One of the main reasons is the binary search’s tendency to produce cache-unfriendly memory access patterns and its relatively high amount of branch mispredictions during the search [LKN13]. Some approaches try to mitigate these effects by using smaller nodes, often as small as a single cache line, which are optimized for cache hierarchies of modern processors [JC10; RR99; SPB05]. However, decreasing the page size down to the size of a single cache line may

be infeasible or at least undesirable. Letting the storage backend handle such small pages would also lead to a considerable overhead. In the end, the choice of a certain page size will always be a trade-off.

We argue, that traditional index-structures for disk-based database systems can be adapted for beyond main-memory database systems. This work focuses on the development of an index structure based on the versatile B-tree layout. Thereby, we try to resolve the previously stated cache-unfriendliness of most B-trees variants. The presented approach hence aims to increase the number of successful cache accesses by applying data access patterns with higher locality. Our approach utilizes a secondary embedded index contained within each page. This secondary index is used to direct incoming searches to narrow down the search space within a given page. Consequently, fewer cache lines will be accessed during the search. We have chosen to retain the B+-tree [Co79] leaf layout, where keys are stored sequentially in accordance to their ordering. This allows us also to maintain the usual strength of B+-trees—their high range scan throughput.

This work’s main contributions are:

- the B²-tree, a disk-based index structure tuned for cache-friendly, page-local lookups,
- the adaption of radix trees to disk-based index structures,
- and a comparison to the already optimized Umbra B+-tree.

The focus of this work lies on the development of an index structure operating on given pages administered by Umbra’s buffer manager. Concurrency is another aspect, our proposal utilizes an optimistic synchronization technique [Ch01], namely Optimistic Lock Coupling (OLC) [LHN19].

This work is structured as follows: Sect. 2 gives a summary of related work on modern index structures. Sect. 3 introduces the B²-tree, which consists of the description of span and decision nodes as well as insertion and retrieval algorithms. Finally, Sect. 4 compares our proposed index structures to Umbra’s B+-tree.

2 Related Work

While there has been constant development and research in the area of index structures, recent approaches mainly focus on main-memory database systems. Many of those index structures are therefore not designed to be used in conjunction with a paging based storage engine, however, their general design may still provide valuable insight.

There are a couple of proposals which aim to improve the cache-friendliness of B-trees. One of which is the Cache Sensitive B+-Tree (CSB+-Tree) [JC10]. Completely different approaches are the so-called Cache-Oblivious B-tree and the Cache-Oblivious string B-tree

[BDF05; BFK06]. Both proposals are based on an important building block, the packed-memory array (PMA). The PMA maintains its elements in physically sorted order, however, elements are not organized in a dense manner, instead, empty spaces will be deployed as necessary [BH07].

The String B-Tree is a B-tree specifically optimized to manage unbounded-length strings [FG99] while minimizing disk I/O. It is composed of Patricia tries [Mo68] as internal nodes where each Patricia trie node stores only the branching character. This architecture enables the use of a constant fanout independent of the lengths of the referenced strings since the Patricia trie leafs only store logical pointers. For this reason, searches within the String B-Tree have to progress optimistically. A search may thereby initially yield a result which does not match the queried key. By comparing the resulting string with the actual query, the length of the longest common prefix will be determined. This information is then used to find the corresponding node within the Patricia trie in question. From there on the correct path based on the actual difference between the resulting string and the queried key will be taken.

Additionally, the choice of a concrete binary search implementation also plays an important role. Index structures which depend heavily on binary search, like B-trees, require an efficient implementation thereof to achieve the best possible performance. Khuong and Morin suggest the use of their branch-free binary search implementation for arrays smaller than the size of the L2 cache [KM17].

Masstree is another key-value store that has mainly been designed to provide fast operations on symmetric multiprocessing (SMP) architectures [MKM12]. It stores all data in main memory, hence it is constructed to be used within the context of main-memory database system. Masstree's design resembles a trie [Br59; Fr60] data structure with embedded B+-trees as trie nodes.

3 The B²-tree

The B²-tree is a variation of the classic B-tree, its core structure is based on the B+-tree layout. We extend the existing layout by embedding another tree into each page, as emphasized by the name B²-tree. The term embedded tree refers to this tree structure, it serves the purpose of improving the lookup performance while maintaining minimal impact on the size consumption as well as on the throughput of insert and delete operations. Our implementation also features some commonly known optimization techniques like the derivation of a shortened separator during a split [Ga18; GL01; Gr11].

Other approaches that combine or nest different index structures have already proved their potential. Masstree for instance showed considerable performance improvements [MKM12]. However, Masstree is not designed to be used in conjunction with paging based storage engines. Another point of concern is the direct correlation between the outer trie height and

the indexed data. The inflexible maximum span length of eight bytes may lead to a relatively low utilization and fanout of the lower tree levels when indexing strings, this is usually caused by the sparse distribution of characters found in string keys. This is not unique to Masstree: ART's fanout on lower tree levels also decreases in such usage scenarios [Bi18]. B+-trees on the other hand feature a uniform tree height by design, since the tree height does not depend on the data distribution. Comparison-based index structures such as the B+-tree on the other hand are often outperformed by trie-based indexes in point accesses [Bi18].

Our approach intends to combine the benefits of both worlds, the uniform tree height of B+-trees with the trie-based lookup mechanics, while still featuring a page based architecture. Our trie-based embedded tree on each page serves the purpose of determining a limited search space where the corresponding queried key may reside. However, we still utilize a comparison-based search on these limited subranges. This design aims to improve the general cache-friendliness of the search procedure on each page.

3.1 The Embedded Tree

In the following we will present the inner page layout of our B²-tree, the general outline can be observed in Fig. 2. As already mentioned, the general page organization follows the common B+-tree architecture, hence, payloads are only stored in leaf nodes. Leaf nodes are also interlinked, like it is originally the case in a B+-tree, in order to maintain the high range scan throughput usually achieved by B+-trees.

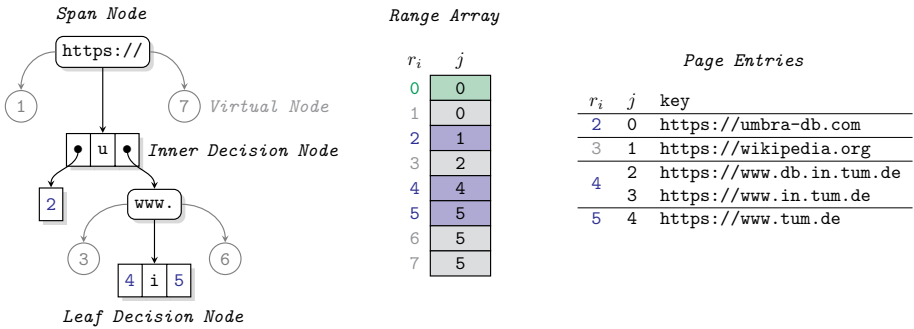


Fig. 2: The embedded tree structure together with an array responsible for translating the values stored in the embedded tree (the r_i) into search ranges where sought key-value pairs may reside. Its values are the exclusive upper bounds of offsets j for the rightmost table (page entries). The grayish virtual nodes are not part of the physically stored tree structure. Empty search ranges are omitted in the rightmost table. This table shows the complete form of the stored keys, without their associated payload.

The embedded tree itself is composed of a couple of different node types. First, we define the *decision node*, it acts like a B-tree node by directing incoming queries onto the corresponding

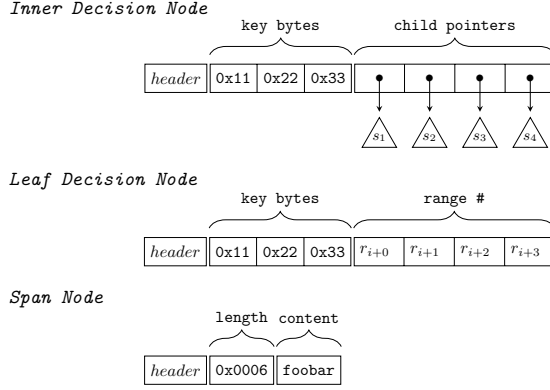


Fig. 3: Memory layout of all the embedded nodes deployed by B²-tree. Each node contains a one byte large header. A flag inside the header determines whether a node contains pointers to subtrees or references to search ranges.

child. Probably the main difference to a B-tree node, is the fact that these nodes operate on a fixed size decision boundary represented by a single byte in our implementation, in contrast to B+-tree nodes, which usually operate on multiple bytes at once. We hence decompose keys into smaller units of information similar to how the trie data structure operates [Br59; Fr60]. Nodes of this decision type direct the search to the first child where the currently investigated byte of the search key is less or equal to the separator. The fanout of this type of node is also limited in order to improve data locality. Another similarity to B-tree nodes is the fact that they can be hierarchically arranged just like B-tree nodes. This node type bears some similarity to the *branch node* found in Patricia tries [Mo68]. However, Patricia’s branch nodes only compare for equality, our decision nodes use the range of bytes to determine the position of a corresponding child. In Fig. 2 this type is illustrated as divided rectangular shape. Fig. 3 illustrates the memory layout for this node type. Note that, inner decision nodes and their leaf counterparts share the same layout, they just differ in the interpretation of their two byte large payloads. Leaf nodes terminate the search for a queried key even if it is not fully processed, the remainder of a queried key will then be further processed by the subsequent comparison based search.

The second node type we define are *span nodes*. These store the byte sequence which forms the longest common prefix found in the subtree rooted at the current node. Their memory layout is shown in Fig. 3. This node type can be compared to the *extension* concept of the Patricia trie [Mo68], however, span nodes have two additional outgoing edges to handle non-equality. Note that, by using an order preserving storage layout for the nodes, there is no necessity to store any next pointer within the span node, since the child node will directly succeed the span node. In Fig. 2 span nodes are illustrated as rounded rectangles. The deployment of span nodes is necessary to advance the queried key past the length of a span if the current subtree has a common prefix. At the following key depth, decisions,

whether a queried key is part of a certain range, can be made once again by the deployment of decision nodes.

Obviously, the content of a span node does not have to match the corresponding excerpt of the queried key exactly. In case the stored span does not match, three scenarios can occur. Firstly, the size of the span may actually exceed the queried key. In that case the input will be logically padded with zero bytes. This may lead to the second case where the input is shorter. Any further comparisons with subsequent nodes are therefore meaningless. Hence, we introduce the concept of a virtual edge pointing from each span to its leftmost child, a so-called *virtual node*. To the edge itself we will refer as *minimum edge*. In Fig. 2 such an edge and its corresponding node is always colored gray to emphasize the aspect that it is not part of the physical tree structure. We follow this edge every time the input is less than the content of the span node. Note that encountering a fully processed input key implies that the minimum edge of a span node has to be taken. Fig. 2 illustrates the usage of this concept with the insertion of the Wikipedia URL after the construction of the embedded tree. This URL does not match the second span node, hence, it is delegated to the virtual node labeled “3”.

The last case where the input is greater than the span node’s content is completely symmetrical to the minimum edge situation. Therefore, a second virtual edge and node pair exists for every span node to handle the *greater than* case. We will cover the algorithmic details more elaborately in Sect. 3.2.

Fig. 2 also illustrates the *range array*, which stores the positions of key-value pairs. These define limited search spaces on the page. This array serves two purposes. First, it eliminates the need to alter the actual contents of the embedded tree during insert and removal operations, this simplifies modification operations significantly. Second, it enables the use of the aforementioned minimum and maximum edges.

During a lookup on a page this array is used to translate the output r_i of a query on the embedded tree into a position j on the actual page. Each lookup on the embedded tree itself yields an index into this array. This array, on the other hand, contains indices into the page indirection vector [Gr06], whereas the indirection vector itself points to any data that does not fit into a slot within the indirection vector [Gr06]. A resulting index thereby specifies an upper limit for the search of a queried key, whereas the directly preceding element specifies the lower limit. In Fig. 2 the annotated positions are colored differently in accordance to their origin. The very first position is colored green, this special element ensures that the lower limit for a search can always be determined. Indices originating from virtual edges are colored gray, whereas blue is used for regular positions. We denote these indices as r_i where i represents the corresponding position within the array of prefix sums. Each r_i occupies two bytes within each leaf node, the memory layout is illustrated in Fig. 3.

Insertion and removal operations, which are to be performed on the overlying page, also affect the embedded tree. More precisely, this affects the search range given by the embedded

tree where the actual operation took place and all subsequent search ranges, since adjusting an upper boundary of one particular search range also implies that subsequent search ranges have to be shifted in order to retain their original limits. This is achieved by simply adjusting the values within the range array for the directly affected search range and every following search range.

3.1.1 Construction

One aspect we have not covered so far is the construction of the embedded tree structure. The construction routine is triggered each time a page is split or merged and also periodically depending on the number of changes since the last invocation.

The construction routine always starts by determining the longest common prefix of the given range of entries beginning at the very first byte of each entry. We will refer to the position of the currently investigated byte as *key depth*, which is zero within the context of the first invocation. On the first invocation, this spans the entire range entries on the current page. Based on the length of the longest common prefix a root node will be created. If the length of the longest common prefix is zero, a decision node will be created, else a span node. In the latter case, the newly created node contains the string forming the longest common prefix. Afterwards, the construction routine recurses by increasing the key depth to shift the observation point past the length of the longest common prefix.

The creation of a decision node is more involved, here we investigate the byte at the current key depth of the key in the middle of the given range. Subsequently, with the concrete value of this byte, a search on the entries right to that key is performed. This search determines the lower bound key index with regard to that value at the current key depth. In some cases, the resulting index may lie right at the upper limit of the given key range. For this reason, we also search in the opposite direction and take the index which divides the provided range of keys more evenly. This procedure is repeated on both resulting subranges until either the size of a subranges falls below a certain threshold or until the physical node structure of the current decision node does not contain enough space to accommodate another entry. Once a decision node is constructed, the construction routine recurses on each subrange, however, this time the key depth remains unchanged. This process is repeated until each final subrange is at most as large as our threshold value.

3.2 Key Lookup

On the page level, the general lookup principle is performed as in a regular B+-tree. The only difference is the applied search procedure. We start by querying the embedded tree which yields an upper limit for the search on the page records within the indirection vector. With the upper limit known, the lower limit can be obtained by fetching the previous entry

from the range array. Afterwards, a regular binary search on the limited range of entries will be performed.

Querying the embedded tree not only yields the search range but also further information about the queried key's relationship to the largest common prefix prevailing in the resulting search range. The concrete relationship is encoded in *skip*, the stored value corresponds to the length of the largest common prefix within the returned search range. It also indicates that the key's prefix is equivalent to this largest common prefix. This information can be exhibited to optimize the subsequent search procedure by only comparing the suffixes.

Algorithm 1 Traversal of the embedded tree structure.

```

1: function TRAVERSE(node, key, length, skip)
2:   if ISSPAN(node) then
3:     (exh, diff)  $\leftarrow$  CMPSPAN(node, key, length, skip)
4:     if diff > 0 then
5:       return MAXIMUMLEAF(node)
6:     else if diff < 0 or exh then
7:       return MINIMUMLEAF(node)
8:     else
9:       spanLength  $\leftarrow$  GETSPANLENGTH(node)
10:      key  $\leftarrow$  key + spanLength
11:      length  $\leftarrow$  length - spanLength
12:      skip  $\leftarrow$  skip + spanLength
13:      TRAVERSE(child, key, length, skip)
14:    end if
15:  else
16:    child  $\leftarrow$  GETCHILD(node, key, length)
17:    if ISLEAF(node) then
18:      return (child, skip)
19:    else
20:      TRAVERSE(child, key, length, skip)
21:    end if
22:  end if
23: end function

```

Algorithm 1 depicts a recursive formulation of the embedded tree traversal algorithm. It inspects each incoming node whether it is a span node or not. We compare the stored span with the corresponding key excerpt at the position defined by *skip*, in case a span node is encountered. The difference between the stored span and the key excerpt will be the result of this comparison. We also determine whether the key is fully processed in this step, meaning that the byte sequence stored within the span node exceeds the remaining input key. Three cases have to be differentiated at this point.

Firstly, the obtained difference stored in *diff* may be greater than zero, hence, the span did not match. However, this also implies that the remaining subtree cannot be evaluated for this particular input key. One of the outgoing virtual edges must therefore be taken. Implementation-wise, this edge is realized by a call to *MaximumLeaf*. It traverses the

remaining subtree by choosing the edges corresponding to the largest values. The final result is thus the rightmost node of the remaining subtree.

The second case, where the excerpt of the input key is smaller, is mostly analog. However, the condition must now not only include the result, whether `diff` is smaller than zero, but also the result, whether the input key has been fully processed during the span comparison or not. An input key that is shorter than the sum of all span nodes, which led to the key's destination search range, will be logically padded with zeros. This leads to another interesting observation. Consider two keys with different lengths and their largest common prefix being the complete first key, all remaining bytes of the second key are set to zero. The index structure has to be able to handle both keys. However, from the point of view of the embedded tree, both keys will be considered as equal. This also implies that the embedded structure has to ensure that both keys will be mapped into the same search range. It is therefore up to the construction procedure to handle such situations accordingly. The subsequent binary search has to handle everything from there on.

The third and last case, where the key excerpt matches the span node, should be the usual outcome for most input keys. We obviously have to account for the actual length of the span to advance the queried key beyond this byte sequence. Hence, the point of observation on the key has to be shifted accordingly. This is also the case where `skip` is adjusted accordingly. It holds the accumulated length of all span nodes which were encountered during the lookup, or an invalid value if one of the span nodes did not match or more precisely if `diff` evaluated to a non-zero value. The subsequent call to either `MaximumLeaf` or `MinimumLeaf` thereupon returns an invalid value for the `skip` entry in the result tuple.

3.3 Key Insertion

We have already briefly discussed, how the insertion of new entries, affects the embedded tree, and its yielded results. Two cases have to be addressed. Either there is enough free space on the affected page to accommodate the insertion of a new entry, or the space does not suffice. A new entry can be inserted as usual if the page has enough free space left. However, this will also require some value adjustments within the range array in order to reflect the change. The latter case, where the page does not hold enough free space for the new entry, will lead to a page split. Splitting a page additionally results in roughly half of the embedded tree being obsolete.

For a simple insertion that does not lead to a page split, updating the embedded tree is trivial. We first determine the affected r_i in the range array where the insertion takes place. The updated search range is then defined by the preceding value and the value at r_i , which has to be incremented, since the search range grew by exactly one entry. In Fig. 2 these index values are denoted as j , and they are stored within the range array. However, this change must also be reflected in all subsequent search ranges. Therefore, all the following entries within the range array have to be incremented as well, in order to point to their original

elements. By conducting this change, subsequent index values will then span all the original search spaces, which were valid up to the point where the insertion occurred.

The case where an insertion triggers a page split has to be handled differently. A split usually implies that approximately half of the embedded tree represents the entries on the original page whereas the other half would represent the entries on the newly created page. Consequently, the index values defining the search ranges of one page are now obsolete. Although, the structure could be updated to correctly represent the new state of both pages, we instead opted to reconstruct the embedded trees. This allows us to utilize the embedded structure to a higher degree, since the current prevailing state of both pages can be captured more accurately. Having a newly split page also ensures that roughly half of the available space is used. We can thus construct a more efficient embedded tree, which specifies smaller search ranges. In turn, smaller ranges can be used to direct incoming searches more efficiently.

3.4 Key Deletion

Deletion is handled mostly analogously. However, the repeated deletion of entries, which define the border between two ranges, may lead to empty ranges. This is no issue per se: The subsequently executed search routine just has to handle such a scenario accordingly. As it is the case with insertions, the deletion of entries also requires further actions. Directly affected search ranges have to be resized accordingly. Hence, the corresponding j values within the range array have to be decremented in order to reflect those changes. All subsequent values also have to be decremented in order to point to their original elements on the page.

3.5 Space Requirements

Another interesting aspect is the space requirement of the embedded tree structure. In the following we will analyze the worst-case space consumption in that regard. We start by determining an upper bound for the space consumption of a path through the embedded tree to its corresponding section of the page which defines a search range.

For now, we only consider the space required by the structure itself, not the contents of span nodes. The complete length of all the contents of span nodes forms the longest common prefix of a certain page section, which our second part of this analysis takes into account. Furthermore, a node in the context of the following first part refers to a compound construction of a decision node and a zero-length span node, this represents the worst-case space consumption scenario, where each decision node is followed by a span. Similar to the analysis of ART's worst-case space consumption per key [LKN13], a space budget $b(n)$ in byte for each node n is defined. This budget has to accommodate the size required by the embedded tree to encode the path to that section. x denotes the worst-case space

consumption for a path through the embedded tree in byte. The total budget for a tree is recursively given by the sum of the budgets of its children minus the fixed space $s(n)$ required for the node itself. Formally, the budget for a tree rooted in n , can be defined as

$$b(n) = \begin{cases} x & \text{isTerminal}(n) \\ \sum_{c \in \text{children}(n)} b(c) - s(n) & \text{else.} \end{cases}$$

Hypothesis: $\forall n : b(n) \geq x$.

Proof. Let $b(n) \geq x$. We give a proof by induction over the length of a path through the tree.

Base case: The base case for the terminal node n , i. e. a page section, is trivially fulfilled since $b(n) = x$.

Inductive step:

$$\begin{aligned} b(n) &= \sum_{c \in \text{children}(n)} b(c) - s(n) \\ &\geq b(c_1) + b(c_2) - x && \text{(a node has at least two children)} \\ &\geq 2x - x = x && \text{(induction hypothesis).} \end{aligned}$$

Conclusion: Since both cases have been proved as true, by mathematical induction the statement $b(n) \geq x$ holds for every node n . \square

An upper bound for the payload of the span nodes is obtained by assigning the complete size of the prefix of each section to the section itself. Assigning the complete prefix directly to a section implies that the embedded tree does not use snippets of the complete prefix for multiple sections, therefore, each span node has a direct correlation with a search range defined by the embedded tree. The absence of shared span nodes, thus, maximizes the space consumption for the embedded tree. An upper bound for the space consumption of the embedded tree is given by

$$\sum_{r \in \text{searchRanges}(p)} (l(r) + x)$$

where $l(r)$ yields the size of the longest common prefix of the search range r within page p .

We can therefore conclude that the additional space required by the embedded tree mostly depends on the choice of how many search ranges are created and the size of common prefixes within them. Our choice of roughly 32 elements per search range yielded the optimal result on all tested datasets, however, this is a parameter which may require further tuning in different scenarios. In our setting, the space consumption of the embedded structure never exceed 0.5 percent of the page. Note that, the prefix of each key within the same search range does not have to be stored, the B²-tree may therefore also be used to compress the stored keys.

In the following we will analyze how modern CPUs may benefit from B²-tree's architecture. Both AMD's and Intel's current x86 lineup feature L1 data caches with a size of 32 KiB,

8-way associativity, and 64-byte cache-lines. Our previous worst-case space consumption showed that the size of the embedded tree is mostly influenced by the size of common prefixes. The constant parameter x , on the other hand, can be set to 15, which is the size of a decision node and an empty span node. With the aforementioned setup of 32 elements per search range and a page size of 64 KiB, we can assume that the embedded structure, excluding span nodes, fits into a couple of cache lines, our evaluation also supports this assumption.

Efficient lookups within the limited search ranges are the second important objective of our approach. With the indirection vector being the entry point for the subsequent binary search, it is beneficial to prefetch most of the accessed slots. In our implementation, each slot within the indirection vector occupies exactly 12 bytes. Therefore, with 32 elements per search range, only six cache-lines are required to accommodate the entire section of the indirection vector. Recall that it is a common optimization strategy to store the prefix of a key within the indirection vector as unsigned integer variable. The B²-tree, however, utilizes this space to store a substring of each key since the prefixes are already part of the embedded tree. We will refer to this substring as *infix*. It can also be observed that the stored infix values within the indirection vector are usually more decisive, since the embedded tree already confirmed the equality for all the prefix bytes. Overall, this implies that fewer indirection steps, to fetch the remainder of a key, have to be taken.

3.6 Concurrency

B²-tree was designed with concurrent access via optimistic latching approaches taken into consideration. While this approach adapts well to most vanilla B-tree implementations, other architectures may require additional logic. This section covers all necessary adaptations and changes required by the B²-tree in order to ensure correctness in the presence of concurrent accesses.

Optimistic latching approaches often require additional checks in order to guarantee thread safety. Leis et al. [LHN19] list two issues that may arise through the use of speculatively locking techniques such as OLC. The first aspect concerns the validity of memory accesses. Any pointer obtained during a speculative read may point to an invalid address due to concurrent write operations to the pointer's source. Readers have hence to ensure that the read pointer value was obtained through a safe state. This issue can be prevented by the introduction of additional validation checks. Before accessing the address of a speculatively obtained pointer, the reader has to compare its stored lock version with the version currently stored within the node. Any information obtained before the validation has to be considered as invalid if those versions differ. Usually, an operation will be restarted upon encountering such a situation.

Secondly, algorithms have to be designed in a manner that their termination is guaranteed under the presence of write operations performed by interleaving threads. Leis et al. discuss

one potential issue concerning the intra-node binary search implementation as such. They note that its design has to ensure that the search loop can terminate under the presence of concurrent writes [LHN19]. Optimistically operating algorithms, therefore, have to ensure that no accesses without any validation to speculatively obtained pointers are performed and that termination under the presence of concurrent writes is guaranteed.

However, the presented traversal algorithm does not guarantee termination without the introduction of further logic. One main aspect concerns the observation that span nodes can contain arbitrary byte sequences. It is hence possible to construct a key containing a byte sequence that resembles a valid node. Such a node may also contain links pointing to itself. An incoming searcher may then end up in a cycle due to previous modifications performed by an interleaving writer which had conducted modifications to the embedded structure in said manner.

To prevent issues such as the one described, certain countermeasures have to be taken. We have to ensure that the traversal progresses with every new node. Furthermore, node pointers must not exceed the boundary of their containing page. We could have used the validation scheme presented by Leis et al. [LHN19]. This would require a validation on the optimistic lock's version after each node fetch. However, we can also use the fact, that in our implementation each parent node has a smaller address than any of its children. We furthermore have to ensure that each obtained node pointer lies within the boundary of the current page. Note that any search range obtained through the embedded tree is also a possible candidate leading to invalid reads. We hence have to ensure that each obtained boundary value also lies within the boundary of the currently processed page. Our binary search implementation, which will be performed directly afterwards, trivially fulfills the previously described termination requirement.

Insert and delete operations do not require any further validation steps, since they do not depend on any unvalidated speculative reads and exclusive locks will be held during such operations anyway.

4 Evaluation

In the following we evaluate various aspects of our B^2 -tree and compare them to the Umbra B+-tree. Note that the Umbra B+-tree is our only reference due to the lack of any other page based index structure capable of running on top of Umbra's buffer manager. In the following we analyze B^2 -tree's performance as well as its scalability, the space requirements for the embedded tree, and the time required to construct the embedded tree.

4.1 Experimental Setup

All the following experiments were conducted on an Intel Core i9 7900X CPU at stock frequency paired with 128 GB of DDR4 RAM. Furthermore, index structures do not have

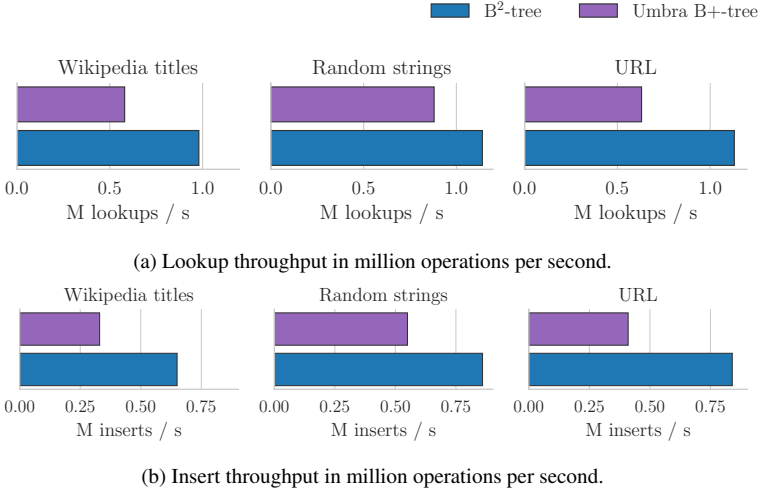


Fig. 4: Single-threaded throughput comparison of the B²-tree and the Umbra B+-tree grouped by the used dataset and imposed workload.

to access background memory, everything will be kept in main memory, unless otherwise stated. B²-tree as well as the standard B+-tree have been compiled to use 64 KiB pages which is the smallest page size Umbra’s buffer manager provides. The evaluation system runs on Linux with GCC 9.3, which has been used to compile all index structures.

Our reference will be the Umbra B+-tree as already stated. This particular B+-tree implementation uses some commonly known optimizations like the choice of the smallest possible separator within the neighborhood of separators around the middle of each page, and a data locality optimization where the first bytes of each key are stored within its corresponding entry in the indirection vector [GL01; Gr06; Gr11].

4.2 Datasets

We have used a couple of different datasets in our evaluation. Those datasets were chosen to resemble real-world workloads to a certain degree. Indexing of URLs and English Wikipedia titles⁶ should resemble real-world scenarios. We also included a completely synthetic dataset consisting of randomly drawn strings, this dataset will be denoted as *Random* dataset.

⁶ <https://dumps.wikimedia.org/enwiki/20190901/enwiki-20190901-all-titles.gz>

	distinct count	average length	median length	min length	max length
Wikipedia	48 454 094	21.82	17	1	257
Random	30 000 000	68.00	68	8	128
URL	6 393 703	62.14	59	14	343

Tab. 1: Parameters of the used datasets.

Tab. 1 summarizes some of the most important characteristics of the used datasets. The Random dataset is generated by a procedure which generates each string by drawing values from two random distributions. Thereby, the first distribution determines the length of the string which is about to be generated. Subsequently, the second distribution is used to draw every single character in sequence until the final destination length is reached.

4.3 Lookup Performance

In the following we will compare the point lookup throughput of our B^2 -tree against our reference. The lookup benchmark queries each key from the randomly shuffled dataset which has been used for the construction of the index itself. Fig. 4a summarizes the results of our string lookup benchmark whereas Fig. 4b shows the influence of B^2 -tree’s more efficient lookup approach onto the insert throughput. B^2 -tree’s lookup throughput is roughly twice as high as that of its direct competitor. Keys in the URL and Wikipedia datasets often share large common prefixes, discriminative bits are therefore often not part of the integer field within the indirection vector of Umbra’s B+-tree. In these situations, the B^2 -tree has an advantage since the entries within the indirection vector are more likely to contain discriminative bits. The Random dataset, on the other hand, features very short common prefixes and a larger amount of discriminative bits between the bit string representation of keys. It is therefore not surprising that the performance gap between the Umbra B+-tree and our B^2 -tree is smaller on this dataset.

	Approach	Inst.	IPC	L1D-Miss	LLC-Miss	BR-Miss
Random	B^2 -tree	1402	0.39	38.32	10.17	15.9
	Umbra B+-tree	2519	0.51	44.63	20.02	19.84
URL	B^2 -tree	1839	0.49	45.69	11.35	22.74
	Umbra B+-tree	3382	0.51	79.58	28.88	16.15
Wikipedia	B^2 -tree	1593	0.38	46.02	13.84	22.76
	Umbra B+-tree	3147	0.43	61.7	30.82	28.22

Tab. 2: Performance counters per lookup operation. The best entry in each case is highlighted in bold type. B^2 -tree mostly dominates the Umbra B+-tree which is in accordance with the previously discussed throughput numbers.

Recording performance counters during experiments usually facilitates further insights, Tab. 2 therefore contains an exhaustive summary. Comparing the averaged amount of instructions required per lookup between the B+-tree and our B²-tree already reveals a considerable advantage in favor of the latter approach. This advantage also exists between the observed amount of L1 data cache misses (L1D-Miss) and last level cache misses (LLC-Miss), where the latter metric reveals that the standard B+-tree produces roughly twice as many misses. This is most likely related to the redesigned search procedure. Thereby, binary search is performed on a smaller search range. Furthermore, the contents of the infix fields within the indirection vector are usually more decisive than the contents wherein stored by the Umbra B+-tree. As a result, the comparison procedure, which will be invoked by the binary search procedure, can often refrain from performing any comparisons on the suffixes stored within the area where the remainder of the records are stored. This also reduces the total amount of cache accesses. For the B²-tree one might expect fewer branch mispredictions, since the infix values are usually more decisive, however, the metric for the amount of mispredicted branches (BR-Miss) per lookup reveals no significant differences between both approaches. This is most likely the result of the additional logic performed during the lookups on the embedded tree.

4.4 Scalability

Additionally, to evaluating B²-tree’s single-threaded point lookup and range scan throughput, we also analyzed its scalability. We ran the same workload as in the single-threaded point lookup experiment. The results of this experiment are shown exemplarily for the URL dataset in Fig. 5. Note that we omitted the results for the remaining datasets due to them being very similar.

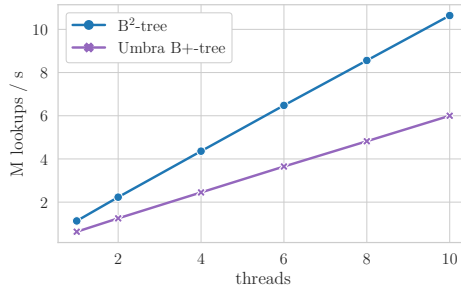


Fig. 5: Scalability on the URL dataset.

Also, the performance difference between our standard B+-tree and B²-tree remains as the number of threads increases. Overall, the B²-tree scales well for still being a B+-tree from an implementation point of view. This also correlates with previous work which did analyze the lookup throughput of B+-trees in combination with OLC [Le18; Wa18].

4.5 Throughput With Page Swapping

The experiment was set up as follows: in the first phase, all the keys of the Wikipedia titles were inserted into an empty index structure. If necessary, pages were swapped out into a temporary in-memory file by the buffer manager. In the second phase, the retrieval time for each key of the randomly shuffled input was measured.

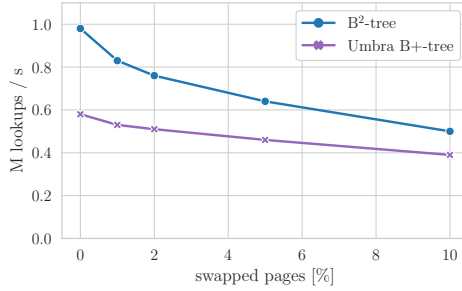


Fig. 6: Lookup throughput on the URL dataset with index structures utilizing Umbra’s buffer manager.

Fig. 6 shows the results of the comparison between these two index structures in dependence of the percentage of swapped out pages. The B²-tree outperforms the Umbra B+-tree for every tested percentage of swapped out pages. However, note that both curves eventually converge as the workloads become increasingly I/O bound.

4.6 Space Consumption

Another important aspect of the presented approach is the total amount of additionally required space on each page. Recall that we use 64 KiB large pages. We were able to fit the complete embedded tree structure in just a couple of hundred bytes as Tab. 3 affirms.

dataset	size [%]
Wikipedia titles	0.48
Random strings	0.49
URLs	0.52

Tab. 3: Averaged space consumption for the complete embedded tree in percent of the page size.

The space utilization of the embedded tree has therefore never been a source of concern in our point of view. However, it should be noted that the size of the embedded tree is variable, and that it will be influenced by the structure of the input data. Especially long shared prefixes have an impact on the overall space consumption of the embedded tree.

5 Conclusion

We presented the B²-tree which speeds up lookup operations by embedding an additional tree into each tree node. The B²-tree showed considerable performance improvements in comparison to an optimized B+-tree. This is related to the total number of instructions required per lookup, which in this case is lower than the number required by the Umbra B+-tree. Our B²-tree, therefore, provides considerable improvements regarding the point lookup throughput. The overhead inflicted by the construction of an embedded tree during each page split is no point of concern as our experimental analysis showed. Furthermore, the additional space required for the embedded structure is mostly negligible, as our evaluation confirmed.

References

- [BDF05] Bender, M. A.; Demaine, E. D.; Farach-Colton, M.: Cache-Oblivious B-Trees. *SIAM J. Comput.* 35/2, pp. 341–358, 2005.
- [BFK06] Bender, M. A.; Farach-Colton, M.; Kuzmaul, B. C.: Cache-oblivious string B-trees. In: *Proceedings of SIGMOD 2006*. Pp. 233–242, 2006.
- [BH07] Bender, M. A.; Hu, H.: An adaptive packed-memory array. *ACM Trans. Database Syst.* 32/4, p. 26, 2007.
- [Bi18] Binna, R.; Zangerle, E.; Pichl, M.; Specht, G.; Leis, V.: HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In: *Proceedings of SIGMOD 2018*. Pp. 521–534, 2018.
- [Br59] Briandais, R. D. L.: File searching using variable length keys. In: *Papers presented at the the March 3-5, 1959, western joint computer conference*. Pp. 295–298, 1959.
- [Ch01] Cha, S. K.; Hwang, S.; Kim, K.; Kwon, K.: Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In: *Proceedings of VLDB 2001*. Pp. 181–190, 2001.
- [Co79] Comer, D.: Ubiquitous B-Tree. *ACM Comput. Surv.* 11/2, pp. 121–137, June 1979.
- [FG99] Ferragina, P.; Grossi, R.: The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *J. ACM* 46/2, pp. 236–280, 1999.
- [Fr60] Fredkin, E.: Trie memory. In: *CACM*. 1960.
- [Ga18] Galakatos, A.; Markovitch, M.; Binnig, C.; Fonseca, R.; Kraska, T.: A-Tree: A Bounded Approximate Index Structure. *CoRR* abs/1801.10207/, 2018.
- [GL01] Graefe, G.; Larson, P.-Å.: B-Tree Indexes and CPU Caches. In (Georgakopoulos, D.; Buchmann, A., eds.): *IEEE Data Eng.* 2001. Pp. 349–358, 2001.

- [Gr06] Graefe, G.: B-tree indexes, interpolation search, and skew. In: Workshop on Data Management on New Hardware, DaMoN 2006. P. 5, 2006.
- [Gr11] Graefe, G.: Modern B-Tree Techniques. Found. Trends Databases 3/4, pp. 203–402, 2011.
- [GUW09] Garcia-Molina, J. W. H.; Ullman, J. D.; Widom, J.: DATABASE SYSTEMS The Complete Book Second Edition.” 2009.
- [Ha08] Harizopoulos, S.; Abadi, D. J.; Madden, S.; Stonebraker, M.: OLTP through the looking glass, and what we found there. In: Proceedings of SIGMOD 2008. Pp. 981–992, 2008.
- [JC10] Jin, R.; Chung, T.-S.: Node Compression Techniques Based on Cache-Sensitive B+-Tree. In: 9th IEEE/ACIS ICIS 2010. Pp. 133–138, 2010.
- [KM17] Khuong, P.-V.; Morin, P.: Array Layouts for Comparison-Based Searching. ACM Journal of Experimental Algorithmics 22/, 2017.
- [KN11] Kemper, A.; Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proceedings of ICDE 2013. Pp. 195–206, 2011.
- [Le18] Leis, V.; Haubenschild, M.; Kemper, A.; Neumann, T.: LeanStore: In-Memory Data Management beyond Main Memory. In: Proceedings of ICDE 2018. Pp. 185–196, 2018.
- [LHN19] Leis, V.; Haubenschild, M.; Neumann, T.: Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. IEEE Data Eng. Bull. 42/1, pp. 73–84, 2019.
- [LKN13] Leis, V.; Kemper, A.; Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: Proceedings of ICDE 2013. Pp. 38–49, 2013.
- [MKM12] Mao, Y.; Kohler, E.; Morris, R. T.: Cache Craftiness for Fast Multicore Key-value Storage. In: Proceedings of EuroSys 2012. Bern, Switzerland, pp. 183–196, 2012.
- [Mo68] Morrison, D. R.: PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. J. ACM 15/4, pp. 514–534, Oct. 1968.
- [NF20] Neumann, T.; Freitag, M. J.: Umbra: A Disk-Based System with In-Memory Performance. In: Proceedings of CIDR 2020. 2020.
- [RR99] Rao, J.; Ross, K. A.: Cache Conscious Indexing for Decision-Support in Main Memory. In: Proceedings of VLDB 1999. Pp. 78–89, 1999.
- [SPB05] Samuel, M. L.; Pedersen, A. U.; Bonnet, P.: Making CSB+-Tree Processor Conscious. In: Workshop on Data Management on New Hardware, DaMoN 2005, Baltimore, Maryland, USA, June 12, 2005. 2005.
- [Wa18] Wang, Z.; Pavlo, A.; Lim, H.; Leis, V.; Zhang, H.; Kaminsky, M.; Andersen, D. G.: Building a Bw-Tree Takes More Than Just Buzz Words. In: Proceedings of SIGMOD 2018. Pp. 473–488, 2018.

Optimized Theta-Join Processing through Candidate Pruning and Workload Distribution

Julian Weise¹, Sebastian Schmidl², Thorsten Papenbrock³

Abstract: The Theta-Join is a powerful operation to connect tuples of different relational tables based on arbitrary conditions. The operation is a fundamental requirement for many data-driven use cases, such as data cleaning, consistency checking, and hypothesis testing. However, processing theta-joins without equality predicates is an expensive operation, because basically all database management systems (DBMSs) translate theta-joins into a Cartesian product with a post-filter for non-matching tuple pairs. This seems to be necessary, because most join optimization techniques, such as indexing, hashing, bloom-filters, or sorting, do not work for theta-joins with combinations of inequality predicates based on $<$, \leq , \neq , \geq , $>$.

In this paper, we therefore study and evaluate optimization approaches for the efficient execution of theta-joins. More specifically, we propose a theta-join algorithm that exploits the high selectivity of theta-joins to prune most join candidates early; the algorithm also parallelizes and distributes the processing (over CPU cores and compute nodes, respectively) for scalable query processing. The algorithm is baked into our distributed in-memory database system prototype A²DB. Our evaluation on various real-world and synthetic datasets shows that A²DB significantly outperforms existing single-machine DBMSs including PostgreSQL and distributed data processing systems, such as Apache SparkSQL, in processing highly selective theta-join queries.

Keywords: theta-join; query optimization; distributed computing; actor programming

1 Theta-Join Processing

A *join* is a powerful operation in relational database theory that allows us to combine tuples of the same or different relational instances. The most popular join operator is the *equi-join* \bowtie that combines tuples based on the equality of certain attribute values. The equi-join serves most basic tuple combination scenarios, such as tuple reconstruction in normalized schemata, knowledge enrichment via data integration, and the resolution of foreign-key relationships. The *theta-join* \bowtie_{Θ} is a generalized join variant that combines tuples based on arbitrary join conditions Θ including but not limited to value equality. A join condition is a boolean statement on the attribute values of two tuples. Following related work, we express any such statement as a conjunction of predicates based on $<$, \leq , \neq , \geq , $>$.

¹ Hasso Plattner Institute for Digital Engineering gGmbH, University of Potsdam, Information Systems, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany julian.weise@hpi-alumni.de

² Hasso Plattner Institute for Digital Engineering gGmbH, University of Potsdam, Information Systems, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany sebastian.schmidl@hpi.de

³ Hasso Plattner Institute for Digital Engineering gGmbH, University of Potsdam, Information Systems, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany thorsten.papenbrock@hpi.de

Geo-spatial querying. Theta-joins are used whenever tuples need to be paired up in a specific way. In *temporal* or *geo-spatial querying*, for example, tuples often need to match in certain value ranges. The query "Combine all *City*-tuples with those *State*-tuples in which they are geographically located"(see Figure 1), for instance, matches tuples based on longitude (*Long*) and latitude (*Lat*) range information.

$$\rho_C(City) \bowtie_{S.Long_{min} \leq C.Long \wedge S.Long_{max} \geq C.Long \wedge S.Lat_{min} \leq C.Lat \wedge S.Lat_{max} \geq C.Lat} \rho_S(State)$$

Fig. 1: A geo-spatial query that builds tuples of cities and their corresponding states.

Data cleaning. Another important area of application for theta-joins is rule-based *data cleaning*. Given a data quality rule or an integrity constraint, such as "*TaxPayers* with higher income need to pay more taxes than *TaxPayers* with lower income", we can formulate the negated constraint as a theta-join query (see Figure 2) to retrieve all data inconsistencies w. r. t. this constraint from the data and clean them afterwards.

$$\rho_{T1}(TaxPayer) \bowtie_{T1.Income < T2.Income \wedge T1.TaxRate > T2.TaxRate} \rho_{T2}(TaxPayer)$$

Fig. 2: A data cleaning query retrieving all inconsistent tuple pairs w. r. t. a given integrity constraint.

Hypothesis testing. The use of theta-joins in the area of *hypothesis testing* works very similar to the data cleaning use case: Given a hypothesis statement, such as "*Countries* that invest more in education have less child poverty and a higher educational level than *Countries* that invest less in education", we can query all pairs of countries that contradict this statement via a theta-join (see Figure 3).

$$\rho_A(Country) \bowtie_{A.ESpend > B.ESpend \wedge (A.CPov \geq B.CPov \vee A.ELevel \leq B.ELevel)} \rho_B(Country)$$

Fig. 3: A hypothesis testing query that collects contradicting tuple pairs.

Hypothesis and integrity statements are often formulated manually by domain experts. They can, however, also be discovered automatically with modern *data mining* and *data profiling* algorithms. Functional dependencies, order dependencies and denial constraints are only a few types of statements that can meanwhile be retrieved automatically [AGN15]. While it has been shown that the mined statements are useful for tasks, such as consistency checking and data cleaning [Bo07; Co17], the amount and complexity of the statements puts significant pressure onto the theta-join operations used for their evaluation.

Although the theta-join is an essential part of relational algebra, its common physical implementation in data query engines is a nested-loop join, i. e., the Cartesian product, in combination with a filter operation. Consider for example the theta-join shown in SQL Query 1, which targets the San Francisco Employee Compensation dataset⁴.

⁴ <https://data.sfgov.org/City-Management-and-Ethics/Employee-Compensation/88g8-5mnd> (08-August-2020)


```

SELECT a.eID, b.eID
FROM EmployeeCompensation a, EmployeeCompensation b
WHERE a.jobCode = b.jobCode
      AND a.salaries < b.salaries
      AND a.eID != b.eID

```

SQL Query 1: Identify pairs of employees performing the same job but being paid differently.

We executed the query on different query engines including *PostgreSQL*⁵, *SparkSQL*⁶ and *Amazon Redshift*⁷, which all produced a query plan similar to the one shown in Listing 2: an equi-join on the equality predicates followed by a post-filter on the inequality predicates.

```

Merge Join
  Merge Cond: (a.jobcode = b.jobcode)
  Join Filter: ((a.salaries < b.salaries) AND (a.ID <> b.ID))
  -> Sort: Sort Key: a.jobcode
      -> Seq Scan on EmployeeCompensation a
  -> Materialize
      -> Sort: Sort Key: b.jobcode
          -> Seq Scan on EmployeeCompensation b

```

Listing 2: The PostgreSQL query plan for SQL Query 1.

With the high selectivity of the equi-join, the query offers a relatively good performance. However, considering the queries of the use cases discussed above, most of them do not employ equality operators. So replacing the equality operator in Query 1 with an inequality operator causes the query engines to produce queries plans similar to the one shown in Listing 3: a nested-loop join with a large post-filter.

```

Nested Loop
  Join Filter: ((a.jobcode <> b.jobcode)
AND (a.salaries < b.salaries) AND (a.ID <> b.ID))
  -> Seq Scan on EmployeeCompensation a
  -> Materialize
      -> Seq Scan on EmployeeCompensation b

```

Listing 3: PostgreSQL Query Plan for SQL Query 1 without equality operator

Because the established hashing-, indexing- and sorting-based optimizations are not applicable for complex join conditions with inequality predicates, the systems fall back to the quadratic comparison of all tuples without employing any optimization. The performance of the nested-loop join in all systems is, therefore, dramatically worse than the performance

⁵ <https://www.postgresql.org/> (08-August-2020)

⁶ <https://spark.apache.org/sql/> (08-August-2020)

⁷ <https://aws.amazon.com/redshift/> (08-August-2020)

of an equi-join. This is problematic when executing theta-joins on real-world datasets as the costs scale quadratically with the datasets' size.

Although there is probably no solution for the quadratic complexity of theta-joins, we can still optimize the join performance by *pruning join candidates* (and, hence, their join condition tests) and by *distributing the join workload* to multiple machines (and, hence, scale out the processing). In this paper, we develop a theta-join algorithm that implements these two optimizations for our distributed in-memory database system prototype A²DB. A²DB is an actor-based and, therefore, inherently parallel and distributable database, which is designed for analytical query workloads. It builds upon the *actor model*, which is a reactive programming paradigm that uses actors as its universal computational primitives. An actor is essentially an object with strictly private state that communicates with other actors using asynchronous messaging. The architecture of this system follows the idea of an actor database system [Be18; SSP19], in which all database state is encapsulated in actors. Our database prototype and, hence, also our theta-join algorithm are implemented using the *akka toolkit*⁸, which is the most popular actor model implementation for the Java Virtual Machine.

Join candidates pruning. Our first optimization is based on the observation that theta-join results in real-world use cases are small (often even empty) and grow rather linearly with the size of the data: Geo-spatial queries result in manageable overlaps, data cleaning queries should return relatively few data quality issues, and hypothesis checking queries are expected to return empty results (or very small results if the hypothesis is not quite correct). For this reason, most real-world theta-joins have a high selectivity. In this paper, we propose a theta-join algorithm that calculates and evaluates the selectivity of the individual join predicates; selective predicates are, then, used to prune the candidate space.

Join workload distribution. Because theta-join results have a quadratic worst-case size in the length of the input dataset, the candidate pruning effects are not always sufficient to process larger join queries. For this reason, our theta-join algorithm facilitates parallelization and can be scaled out to multiple compute nodes. The ability to scale also naturally exploits the distributed storage of data in the A²DB system.

In the following, we first discuss related work in the area of (theta-)join processing and the limitations of existing approaches (Section 2). We then explain how data is maintained and distributed (Section 3). With these basic details explained, we first describe our distributed theta-join algorithm (Section 4) and then its selectivity-based join strategies (Section 5). In an extensive evaluation, we then compare the performance of our theta-join algorithm with the performance of the data processing systems *PostgreSQL* (single node), *SparkSQL* (12 node cluster) and *Amazon Redshift* (12 node cloud) to demonstrate that A²DB can process selective theta-joins significantly faster than the state-of-the-art Carthesian product plus post-filter approach (Section 6).

⁸ <https://akka.io> (08-August-2020)

2 Related Work

In this section, we give an overview of existing work in the area of theta-join processing. We take a brief look at the origin of join operations in the relational model but focus our investigation on efficient and distributed answering of theta-joins.

Relational Joins were first discussed by Codd in 1970 as a concept of combining tuples based on attribute equality in his proposal for the relational data model. He later extended his proposal by combining tuples also with non-equality operators, which was the introduction of the *theta-join* operator [Co79]. Early subsequent work then mainly focused on the equality-based join operation and suggested various implementations and optimizations to calculate these equi-joins efficiently [Go75]. Prominent examples are hash, sort-merge, and nested-loop joins, as well as techniques utilizing indexes [ME92].

Parallel and Distributed Equi-Join Processing techniques have been examined extensively in response to the development of multi-core machines. Specifically, researchers identified challenges and proposed solutions for typical problems in multi-threaded and distributed systems, such as shared state and workload partitioning [ESW78; VG84]. A prominent optimization for calculating joins in distributed systems, which was proposed by Bernstein, utilizes semi-joins to extract join candidates [BC81]. In this way, the communication overhead and, hence, query processing time could be decreased significantly. Most of the techniques proposed in this area can, however, not be applied to theta-joins, because they do not support complex theta predicates.

Efficient Theta-Join Processing is the goal of the *IE-Join* algorithm by Khayyat et al. [Kh15]. The algorithm applies a sophisticated sort-merge approach by first sorting the values of up to two join attributes and implicitly identifying candidate sets. Via permutation arrays and clever bitset operations, the algorithm tests all predicates of the join condition successively while effectively pruning candidates on the way. In this way, IE-Join is orders of magnitude faster than both PostgreSQL and SparkSQL. Despite its superior performance, the proposed approach is inherently limited to only two join predicates. Adapting IE-Join to more than two predicates requires exactly the strategies proposed in this paper: a strategy to choose the two IE-Join predicates and a post-processing step for all non-chosen predicates.

Distributed Theta-Join Processing optimizations mostly target batch processing and data flow engines, such as *Apache MapReduce* or *Apache Spark*. These engines are effective in processing equi-joins, because distributed grouping and aggregation of tuples is baked into their core feature set, but ineffective for theta-joins, because the grouping does not innately support inequality operators. The *1-Bucket-Theta* algorithm by Okcan et al. [Ok11] is a theta-join processing approach on MapReduce that splits the quadratic comparison space into buckets, which are then processed distributedly by different machines to share the comparison load. *M-Bucket-Theta* enhances the 1-Bucket-Theta algorithm in that the algorithm can detect empty regions in the matrix and prune non-contributing join candidates. Because the algorithm depends on a single comparison matrix, theta-joins with more than

one predicate are handled by concatenating the predicates' attributes into one key. For this reason, M-Bucket-Theta can handle theta-joins with only one predicate effectively. The work of Koumarelas et al. [KNG18] proposes several strategies to optimize M-Bucket-Theta's efficiency for low-selectivity queries. By manipulating the matrix of the mapping phase such that larger regions of it can be pruned, the strategies reduce the algorithm's communication and computation costs by up to 45% and 50% respectively. Despite these performance improvements, the theta-join algorithm still cannot handle multiple predicates. Because multiple-predicate theta-joins are a given for most use cases, such as hypothesis testing and data cleansing, we do not evaluate these approaches in this work.

To join more than two relations at once, Zhang et al. studied the problem of decomposing a multi-way theta-join into multiple binary joins and proposed different strategies and a cost model for optimizing the overall processing time [ZCW12]. For the execution of chained joins, the authors rely on variations of M-Bucket-Theta. In this paper, we consider multi-way theta joins as orthogonal work and focus on efficiently joining two relations.

Besides the MapReduce-based theta-join approaches, Apache Spark supports join processing with arbitrary join conditions innately with its relational module *SparkSQL* [Ar15]. The data flow engine offers *DataFrames* as an abstraction for distributed datasets and an SQL engine to query these datasets. Although the engine also falls back on *Broadcast-Nested-Loop-Joins* when processing theta-joins, the framework is significantly faster than MapReduce.

The capability of distributed theta-join processing can also be found in many commercial DBMSs, such as *Amazon Redshift*. Redshift is a distributed data warehouse solution hosted exclusively in the Amazon Web Services (AWS) cloud. It distributes data across a cluster of configurable size and involves all nodes in query answering. Redshift in particular claims itself to be an efficient and scalable solution for experimenting with (possibly huge) amounts of data [Gu15], which makes it a perfect baseline for our experimental evaluations.

3 Data storage in A²DB

Before we introduce our theta-join algorithm, we need to explain how our database system prototype A²DB stores and handles data. A²DB is an actor-based, distributed in-memory relational DBMS for analytical query workloads that facilitates a leader-follower architecture:

Leader Node: One dedicated node in the A²DB cluster takes the role of a leader. It is responsible for bookkeeping the follower node's membership state, accepting queries and loading data into the database.

Follower node: An A²DB follower is a node in the cluster that is responsible for maintaining and querying portions of the data. Which portions, i. e., partitions of the data a follower is responsible for is defined by the leader node.

Follower nodes play an active role in query processing: The leader node breaks every submitted query, such as a theta-join, into multiple work packages and assigns them to individual follower nodes. Once a follower node receives a work package, it requests

necessary data from other nodes, decides the best local query execution strategy, processes the local results, and sends the results of the query back to the master.

The partitioning of the data in A²DB follows the *PAX* concept: The entire relational dataset is sliced horizontally into equally sized partitions and every partition is stored columnar-wise on one follower node. For every column in a partition, A²DB maintains column-specific metadata, such as the column's *min* and *max* values in this partition and a pre-calculated *sorting* of the partition tuples w. r. t. this column. During query processing, the extreme values can be used to prune this partition and the pre-sortings support sort-based query operators, such as sort-merge joins.

Strictly following the actor programming model, all partitions in A²DB are represented as autonomous actors, which is, in private, non-parallelizable actor state. To access data owned by another actor (e. g., in a join scenario), partition holder actors need to ask other partition holder actors via asynchronous messaging for certain tuples, columns, or values.

4 Theta-Join Workload Distribution

When theta-joining two relations R and S , the query engine needs to validate each possible combination of tuples from both relations against the join condition Θ . Hence, up to $|R| \times |S|$ comparisons need to be performed. Our first approach to efficiently process these comparisons is to distribute the workload to any given number of nodes. When a query is issued, the data is already horizontally partitioned on these nodes. In this section, we propose a reactive join strategy that decomposes and distributes the Θ evaluations. Figure 4 visualizes the general idea of our approach with an example: A²DB splits the join space of two relations R and S and their partitions R_i and S_i into *node-joins*, such as $R_A \bowtie_{\Theta} S_B$, and each node-join then into *partition-joins*, such as $P_{R1} \bowtie_{\Theta} S_{S1}$. A partition-join comprises two partitions P_{Ri} and P_{Si} and constitutes the smallest work-package in the system. In the example, the theta-join consists of four node-joins and each node-join consists of four partition-joins – usually, though, an A²DB cluster consists of more nodes and partitions.

Figure 5 depicts the process of executing a theta-join: When the leader node receives a theta-join query, it creates the node-join matrix that partitions the join into node-joins. It then opens a query session, which causes all follower nodes to calculate their local node-join, i. e., all $R_A \bowtie_{\Theta} S_A$, $R_B \bowtie_{\Theta} S_B$ etc. To perform a partition-join, a processor evaluates all tuple combinations $((t_R, t_S) \mid t_R \in P_{Ri}, t_S \in P_{Sj})$ against the join condition Θ and sends the matching tuples to the result set of the theta-join on the leader. Whenever a follower node finishes a node-join, the leader serves the follower with another node-join. This reactive work pulling mechanism keeps all cluster nodes busy until the join is completed. The leader coordinates this process so that, in the end, all partition-joins are executed. Later in this section, we discuss the leader's node-join selection strategy in more detail.

Every node-join is calculated on one follower node. On that node, the calculation is strongly parallelized and consists of three overlapping steps, which are also shown in Figure 5: The

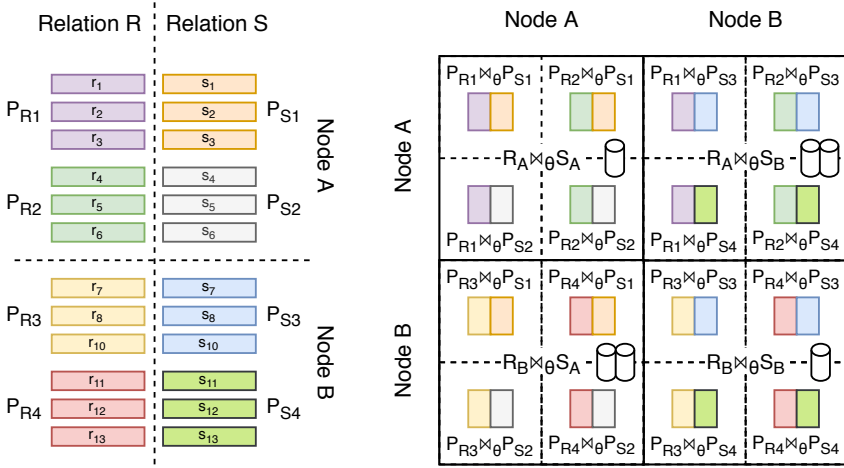


Fig. 4: Distribution of the theta-join calculations over two nodes.

work generation step splits the node-join into partition-join tasks. The *data loading* step then fetches all necessary remote partition data on demand from the other node of the current node-join; the step makes sure that every partition is retrieved only once and it is skipped by the self node-joins, e. g., $R_A \bowtie_{\theta} S_A$. Once a remote partition is available, the *execution* step can start to join this partition with every local partition; every partition-join is executed reactively on one actor and, hence, in parallel to other partition joins. Once all partition-joins are calculated and their results are sent to the leader, the node-join is completed and the follower is ready for the next node-join.

In the following, we discuss the orchestration of the node-joins (Section 4.1), the provisioning of partitions (Section 4.2), and the actual join processing (Section 5).

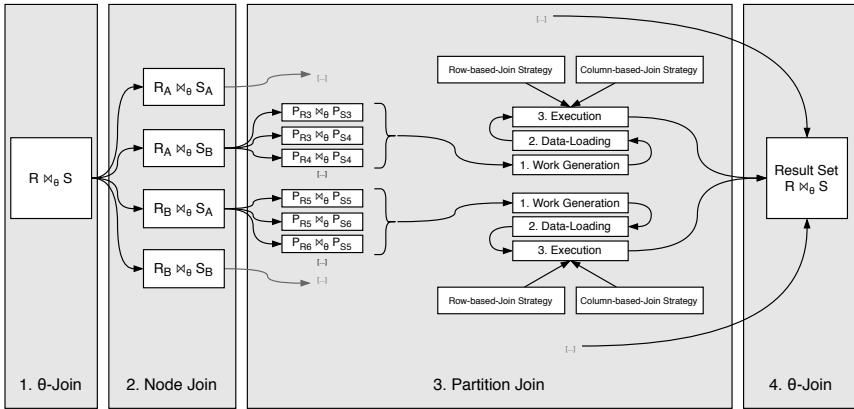


Fig. 5: Overview of the entire distributed theta-join processing process.

4.1 Execution Plan

The *execution plan* is essentially a dynamically created actor on the leader that represents the distributed execution of a theta-join query. It opens a cluster-wide session (involving session actors on all follower nodes), orchestrates the intermediate query processing steps and collects the query result. Through the session actors, all follower nodes know each other and can communicate on a peer-to-peer basis.

With the execution plan, A²DB aims to orchestrate the node-joins in an optimal way. It does so by sending out node-joins as work packages to the followers' query *executors*. The initial task for each follower's executor is to calculate the local node-join. Afterwards, the executors start pulling further node-joins from the execution plan and the task of the execution plan is to serve these requests in a best possible way. This means primarily that every node-join should be handled by a node that owns at least one of the node-join's sides, i. e., $R_A \bowtie_{\Theta} S_B$ should be handled by node A or B . Furthermore, we assign both node-join directions to the same node, i. e., $R_A \bowtie_{\Theta} S_B$ and $S_B \bowtie_{\Theta} R_A$ are one work package that goes to either node A or node B . In this way, partitions are not sent in both directions. However, by assigning the node-joins naively in, for instance, node order, the execution plan quickly encounters requests by a node, whose partitions have all already been joined elsewhere, and therefore cannot serve this node with optimal work. Because followers usually finish their node-joins unevenly fast, the execution plan cannot plan the node-join distribution in advance and, instead, chooses the node-joins reactively based on three heuristics:

1. **Data Locality:** Assign a node-join that involves the partitions of the requesting node, if possible. This rule has the highest priority and overrules all other heuristics.
2. **Selection Flexibility:** Assign a node-join with the least often joined node. By joining the least often joined node next, the execution plan maintains the highest possible flexibility for future join selections – it effectively tries to avoid situations where all node-joins of a particular requesting node are already done. For this, the execution plan counts the number of already assigned node-joins per node.
3. **Query Politeness:** Assign a node-join with the least often requested node. If all potential join partner have the same join counts, selecting the least often requested partner should avoid uneven loads for sending out local partitions. For this heuristic, the execution plan also counts the number of partition requests per node.

Work assisting: Despite these heuristics, the execution plan cannot always meet the first rule, especially at the end of the execution. So if a node cannot be served with a node-join involving itself, the execution plan assigns a node-join according to rule two and three. We refer to the process of taking over foreign node-joins as *work assisting*. The processing of such node-joins requires the execution node to fetch partitions from two nodes instead of one, which is more expensive. Hence, to decide whether work assisting is actually beneficial, the execution plan tracks three additional runtime metrics per node: the average execution time of node-joins $t_{average}$, the execution time for the current node-join $t_{elapsed}$, and the average partition transfer delay t_{delay} . Then, work stealing is done only if the expected

remaining execution time ($t_{average} - t_{elapsed}$) is larger than the expected additional network delay of executing the next node-join as a foreign-node-join (t_{delay}).

Work stealing: If work assisting is no longer possible, follower nodes may support other follower nodes in finalizing their current node-joins by “stealing” some partition-joins. In contrast to work assisting, *work stealing* does not take over an entire node-join but some portion of the remaining partition-joins. For this, the execution plan actor instructs the work requesting follower to steal half of the partition-joins from the follower with the shortest current node-join time $t_{elapsed}$, which should be the follower with the heuristically most unfinished partition-joins. The stealing follower then retrieves these partition-join tasks from the target follower in a peer-to-peer fashion. Both followers report their join results directly to the leader; the leader takes care that no follower is “robbed” more than once.

4.2 Context-specific Partition Provisioning

Before a follower node requests partitions from another follower node, it first exchanges both the join condition Θ and the headers of the involved partitions (see Section 3) with the other follower. The join condition and header metadata help the follower nodes to exchange only required, i. e., context specific partition data. Given the node-join $R_A \bowtie_{\Theta} S_B$, then only a portion of S ’s partitions on node B are relevant for the node-join on node A :

1. A requires only those attribute values from B ’s partitions that are used in Θ . Thanks to the column-oriented format of the partitions, these attributes can easily be selected.
2. A requires only those records from B ’s partitions that intersect with A ’s partitions w. r. t. all of Θ ’s attribute-specific join operators, which are $<, \leq, =, \neq, \geq, >$. The overlap can be checked quickly with the partition’s min and max values of each attribute.

As an example for condition 2, if two partitions P_{Ri} and P_{Sj} have no overlap in attribute x , i. e., $P_{Ri}.x_{min} > P_{Sj}.x_{max}$ and the join condition is $R.x < S.x$ or $R.x \leq S.x$, then A^2DB does not transmit P_{Sj} , because the join of these partitions is empty. If P_{Ri} and P_{Sj} overlap partially, the records are filtered so that the range (min and max) of the transmitted values matches all Θ conditions. In other words, given $R.x < S.x$, node B sends only those local P_{Sj} records where $P_{Sj}.x > P_{Ri}.x_{min}$. With this minimal checking overhead, A^2DB can prune many partition values from the sending process.

5 Theta-Join Candidate Pruning

As already shown in Figure 5, the node-join processing consist of three steps: work generation, data loading, and execution. The work generation splits the node-join into partition-joins and puts the resulting tasks into a task queue. To not overload the memory or network

and to allow other followers to steal work from the task queue, data loading and execution operate on a pull-based execution model: Free worker actors consume partition-join tasks, which first causes missing partition data to be loaded and, once the data arrives, be joined. Via slight over-provisioning and data pre-fetching, the A²DB follower nodes maximize both CPU and network utilization. The final partition-join execution step takes as input the partitions P_{Ri} and P_{Sj} , the partition header metadata, and the join condition Θ .

5.1 Predicate-specific Selectivity Calculation

Before A²DB starts the actual join calculation, it first determines the selectivity of each join predicate $P_{Ri}.x \bowtie_{\vartheta} P_{Sj}.x$ in Θ with $\vartheta \in \{<, \leq, =, \neq, \geq, >\}$. Based on the selectivities, each follower can later choose the best join strategy for its current partition-join. To calculate the selectivity for each join predicate, A²DB exploits the pre-calculated sortings of every attribute in a way that requires at most $2 \cdot (|P_{Ri}| + |P_{Sj}|)$ tuple comparisons. For this, we interpret each predicate as a $|P_{Ri}| \times |P_{Sj}|$ matrix of record pairs. We then draw two lines into this matrix that separate matching tuples from non-matching tuples w. r. t. the predicate's join operator ϑ . Figure 6 shows example results for all operators. To calculate the selectivity, we simply sum up all ranges of matching tuples and divide the result by $|P_{Ri}| \cdot |P_{Sj}|$.

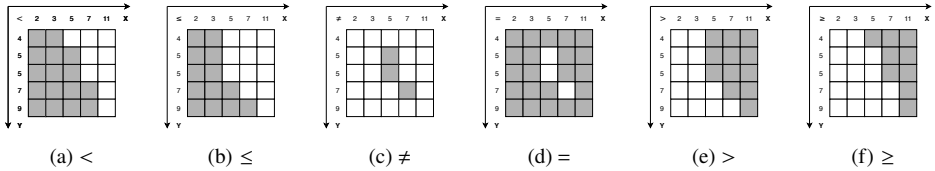


Fig. 6: Exemplified join matrices for all join operators supported by A²DB

To calculate the border lines efficiently, a partition-join worker starts with two pointers l and r in the left-upper corner of the matrix. It then compares the records at the pointer locations and advances the l pointer in a way that it follows the left index of matching tuples and r follows the right index of matching tuples. So for example, if $P_{Ri}(l) \vartheta P_{Sj}(l)$ is true, i. e., the records' values at pointer location l match, l advances downwards; otherwise, it advances to the right. For the same comparison, r advances to the right for matches and downwards, otherwise. The calculation ends when both pointers arrive at the bottom of the matrix. Note that this procedure does not work for \neq , because \neq defines two areas of matching tuples; hence, A²DB calculates \neq as $=$ and inverts the resulting counts. The matching tuple pairs are technically stored in an $|P_{Ri}|$ long array of ranges (see Figure 7). We call this the join *candidate set* for predicate ϑ . The range indexes are given by the l and r pointers whenever these pointers move downwards. The selectivity calculation is executed for all join predicates of Θ in parallel and finishes when all branches are done.

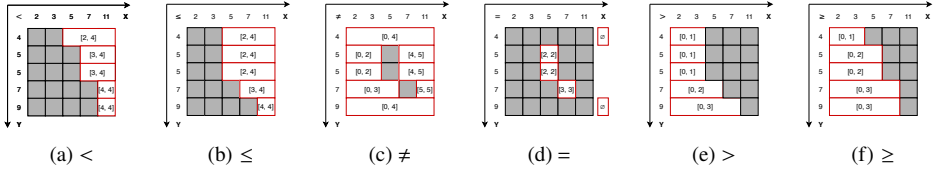


Fig. 7: Representation of matching tuples pairs in ranges

5.2 Partition-Join Strategies

The intersection of all candidate sets, i. e., all tuple pair sets of all predicates, is the theta-join result for $P_{Ri} \bowtie_{\Theta} P_{Sj}$. However, because the join matrices (mostly) use different sortings, the calculated ranges cannot be intersected directly. There are basically two join strategies that we can follow at this point: *row-oriented joining* and *column-oriented joining*.

Row-oriented Join Strategy: The row-oriented joining strategy takes a set of join candidates, which is, for example, the candidate set of one join predicate, and checks each candidate against the entire join condition Θ . This immediately validates the entire join tuple, i. e., one result row. Because Θ is formulated in conjunctive normal form, the predicate testing for one tuple pair stops immediately and discards the pair if a predicate is invalid.

Column-oriented Join Strategy: The column-oriented joining strategy successively intersects the sets of matching tuple pairs of all join predicates. Thereby, the strategy basically tests one predicate after the other vertically for all result tuples, i. e., column-wise for each join attribute. To intersect two candidate sets, which are represented as lists of tuple ranges based on attribute-specific sortings, A²DB first translates both candidate sets into matrices with same tuple sortings in both dimensions. Both resulting matrices are represented as $|P_{Ri}|$ -long array of sparse bitsets (1-bits for matches; 0-bits for no-matches). After this transformation, A²DB can efficiently intersect these candidate sets. The costs for the transformation and intersection depend on the selectivities of the two join predicates.

Both the row- and the column-oriented join strategy profit from considering the most selective predicates, which is the one with the lowest selectivity factor, first: The candidate set of the most selective predicate is the smallest and, hence, prunes the most candidate evaluations when chosen as initial candidate set for row-oriented joining; similarly, intersecting the sparsest candidate sets first in column-oriented joining maximizes the pruning effect of every intersection step and, hence, the overall compute efficiency. For this reason, A²DB sorts the predicates by their selectivity factors and uses the most selective predicates first – regardless of the join strategy, which it needs to choose right after sorting the predicates.

Choosing the row-oriented strategy effectively uses one candidate set for pruning. However, if there is no single predicate with a high selectivity (low selectivity factor) and further predicates are needed to sufficiently prune the candidate space, this strategy alone loses a lot

of pruning potential. On the other hand, choosing the column-oriented strategy exploits all pruning aspects, but because the transformation of range-based into bitset-based candidate sets is expensive, translating all candidate sets outweighs the pruning effect. Considering the examples in Figure 6 shows that, for instance, neither (b) nor (f) are particularly effective, but their intersection, which is (d), is highly selective; the candidate set (c) is neither alone nor in any combination effective enough to compensate its translation costs. For these reasons, we propose a *strategy selection heuristic* that combines both approaches.

Strategy Selection Heuristic: All workers decide the join strategy for their current partition-join task based on the local selectivities and independently of one another. After all predicates are sorted in ascending order according to their selectivity factors, a worker follows the following decision heuristic:

1. **Only one predicate:** If the join condition Θ consists of only one predicate, A²DB simply returns the candidate set of that one predicate, which is the result of the current partition-join.
2. **100% selective:** If the most selective predicate matches no tuple pair, the partition-join result is empty, because the predicate with the highest selectivity defines an upper bound for the number of join results; hence, an empty set is returned.
3. **95 – 100% selective:** If the most selective predicate matches only at most 5%, it alone is so selective that column-oriented joining is not promising. For this reason, the worker uses only the first predicate’s candidate set as input for row-oriented joining.
4. **< 95% selective:** If no highly selective predicate exists, the worker intersects the two most selective predicates via column-oriented joining and feeds the resulting candidates into row-oriented joining.

The last case uses only the first two predicates for column-oriented joining, because we observed that the first two predicates usually have such a strong combined pruning effect that adding a third predicate does not pay off. Different settings of the proposed 95% decision threshold can cause faster execution times depending on various factors, such as dataset size, cluster size and cluster speed, but thresholds around 95% showed the best and also very similar (hence robust) performance results in our experiments. This is due to the generally high selectivity of most real-world theta-join queries and the fact that all workers choose their strategies independently.

Whenever a worker finishes a partition-join, it sends the results to the leader node and fetches the next partition-join from the work queue. After completing all partition-joins, the current node-join is completed and the follower pulls the next node-join from the leader’s execution plan. Once all node-joins are done, the execution plan actor reports the final result to the query issuing client and closes the join session.

6 Evaluation

We now evaluate A²DB's theta-join performance against the theta-join performance of three state-of-the-art data query engines: *PostgreSQL* as a modern representative for a single machine (non-distributed) DBMS, *Amazon Redshift* as a distributed and highly scalable relational DBMS, and *Apache SparkSQL* as a distributed batch-processing system with theta-join capabilities. For the experiments, we configured these systems as follows:

A²DB runs on a 12 node cluster, where each node has an Intel Xeon E5-2630 v4 CPU (20 threads), 32 GiB RAM and 1 GiBit/s Ethernet. The nodes run Ubuntu 18.04.4 and Java 1.8 with G1 garbage collector. A²DB uses a maximum partition size of 5,000 tuples for datasets smaller than 500,000 tuples and a maximum partition size of 10,000 tuples, otherwise. In this way, each node hosts at least one maxed out partition in all evaluations.

PostgreSQL version 10.12 uses one of the nodes described above, but with 64 GiB RAM. We optimized the default configuration of PostgreSQL to achieve a better performance for analytical workloads as suggested in the official documentation⁹: We set the `shared_buffers` to 25% of the system's main memory, which is 16 GiB. The `work_mem` is increased to 512 MB, to not exhaust the memory but still provide enough memory for executing queries mainly in memory. We also increased the `temp_buffers` to 512 MB.

Apache SparkSQL version 2.4.4. uses the same cluster than A²DB. The driver program is written in scala version 2.12 and uses the Hadoop distributed file system as storage technology for the datasets and the query results.

Amazon Redshift needs to be hosted on different hardware in the AWS cloud. Its cluster consists of 12 dc2.large nodes and runs Redshift version 1.0.17498. Each node is an EC2 cloud computing resource with an Intel E5-2686 v4 CPU (two threads), 15 GiB RAM, and 160 GiB NVMe SSDs. To run only one query at a time, we changed the query queue configuration to prohibit concurrent query execution and use all available memory.

Because the hardware for Redshift differs, we define the following rules for comparing the query times of the four systems: A²DB is truly better than PostgreSQL only if it is at least 11 times faster than PostgreSQL (because it has 11 times more nodes); A²DB is better than Redshift only if it is at least 10 times faster (because it has 10 times more threads) and A²DB is clearly slower if Redshift is faster despite its disadvantage – otherwise, we cannot specify which query processing time is better as we do not know Redshift's scalability with the number of local threads (note that Redshift is also highly optimized and specifically tuned for being executed on AWS hardware); A²DB and SparkSQL are directly comparable.

For our experiments, we use synthetic and real-world datasets, which are differently sized subsets of four base recordsets (see Table 1): The synthetic *TPC-H* benchmark dataset, the employee compensation dataset *DataSF* of San Francisco, the US Bureau of Transportation

⁹ <https://www.postgresql.org/docs/10/runtime-config-resource.html> (08-August-2020)

dataset *Flight*, and the extended edited synoptic *Cloud* reports dataset. For the purpose of identifying single rows, we extended all datasets with an additional surrogate key, which is a dedicated, monotonic increasing integer *id* column. We cut down the *Cloud* dataset to five million records, because all systems struggled with its entire size.

Dataset	# Rows	# Columns	Size on disk	Domain	Real-World
TPC-H	6,001,215	25	1,639 MB	Order Management	✗
DataSF	968,373	22	197 MB	Public Administration	✓
Flight	7,268,232	15	701 MB	Flight Control	✓
Cloud	384,584,555	28	521 MB	Weather	✓

Tab. 1: Recordsets used for dataset creation

Our theta-join workload consists of 12 manually crafted theta-join queries. None of the queries contains equality predicates ($=$). Hence, common join algorithms and optimizations do not apply for any of them. Instead, all predicates are based on $<$, \leq , \neq , \geq , $>$. The number of predicates in the queries varies between 2 and 13: TPC-H (5,2), DataSF (2,4,3), Flight (3,4,5,4), and Cloud (5,13,4). We always execute each query with two warm-up executions and report the arithmetic mean of the last five executions. A²DB’s theta-join algorithm, the base recordsets and our theta-join SQL queries are available online¹⁰.

6.1 Equi-Join vs. Theta-Join

To demonstrate the remarkable performance gap between equi-join and theta-join executions, our first experiment compares the performance of an equi-join with the performance of a theta-join. To create the equi-joins for this comparison, we take the two TPC-H queries from our theta-join workload and exchange all their non-equality operators with equality operators. Table 2 shows the measured execution times on the TPC-H dataset.

Recordset	Dataset	Query	Results	PSQL [◇]	SparkSQL	Redshift*	A ² DB
TPC-H	orig.	Q1	0	†	29,345,373	18,630,581	9,371,830
		Q1-Eq	6,366,031	25,616	8,998	332,475	158,315
		Q2	30,980,486	†	24,839,845	31,970,932	440,435
		Q2-Eq	833,567	12,942	5,774	4,483,270	64,198

†: Timeout after 24 hours

*: 2 instead of 20 hyper-threads

◇: 1 instead of 11 nodes

Tab. 2: Query Execution time (in ms) comparison of Equi- vs. Theta-Join

The results show that the equi-joins perform orders of magnitude better than the theta-joins on all systems. This is because the systems can use sophisticated equi-join algorithms, such as (distributed) sort-merge joins, and, therefore, do not need to compare all tuple pairs. Note that the performance gain for equi-joins is not necessarily tied to smaller results: Q1’s result gets larger when being turned into an equi-join, while Q2’s result gets smaller, and in both cases the equi-join is faster. Even though A²DB’s theta-join algorithm is not optimized for

¹⁰ <https://hpi.de/naumann/s/a2db-theta-joins> (30-November-2020)

equality predicates, it also achieves considerably faster execution times for equi-joins. For theta-join query Q2, A²DB is clearly more efficient than all state-of-the-art competitors.

6.2 Query Performance

For a broader performance comparison of A²DB and its competitors, we now measure the query times for all 12 theta-join queries on different subsets of our evaluation datasets. The results of this experiment are listed in Table 3.

The query times show that A²DB significantly outperforms existing single-node DBMSs, such as PostgreSQL, and distributed batch-processing systems, such as Apache Spark, in processing highly selective theta-join queries. PostgreSQL in particular struggles to answer many theta-join queries within 24 hours that A²DB can process in minutes. Considering the setup differences for Redshift and A²DB, which is that Redshift has 10 times fewer threads but also over-optimizes on its hardware, both systems compete quite well. As we know that Redshift does not use join techniques, join operators or join plans optimized for theta-joins, we can conclude that its technical optimizations can actually compete with A²DB's algorithmic optimizations. However, A²DB clearly outperforms Redshift on some queries, such as TPC-H-Q2, Flight-Q1, and Flight-Q4, which are particularly selective. A²DB's selectivity calculations for the individual predicates comes at the expense of extra processing time (e. g., Cloud-Q2), but the overhead is usually negligible w. r. t. the high and quadratic tuple matching costs (e. g., TPC-H-Q2).

A²DB performs particularly well on TPC-H-Q2, Flight-Q1 and Flight-Q4, because it successfully identifies the most selective predicate, e. g., `p_retailprice >= l_extendedprice` for TPC-H-Q2 with a selectivity factor of about 1%, and prunes the candidates accordingly. On most other queries, such as SF-Q2, no single, super selective predicate exists and A²DB needs to combine predicates for candidate pruning. With 13 predicates, query Cloud-Q2 is the largest query in the workload. Interestingly, neither PostgreSQL nor Redshift show a significant performance difference on Cloud-Q2 compared to the other queries; A²DB is more affected by this high number of predicates, because the join matrix calculations take a larger share of the entire query processing time. In summary, A²DB performs best on highly selective theta-join queries with possibly few predicates, which is exactly the kind of theta-join query that we observe in most use cases.

6.3 Scaling Follower Nodes

To utilize all available resources for query answering, A²DB parallelizes and distributes the theta-join processing across a cluster of compute nodes. We now evaluate A²DB's horizontal scalability by measuring the query execution time for both TPC-H queries on an increasing number of follower nodes to evaluate the effectiveness of the distribution. The

Dataset	Subset	Query	PostgreSQL [◇]	SparkSQL	Redshift*	A ² DB
TPC-H	100k	TPC-H-Q1	2,392,800 \pm 1.2%	87,587 \pm 02.7%	4,770 \pm 00.9%	4,497 \pm 04.7%
		TPC-H-Q2	2,107,905 \pm 1.6%	79,581 \pm 32.5%	9,038 \pm 07.8%	248 \pm 15.7%
	500k	TPC-H-Q1	56,685,284 \pm 0.0%	231,617 \pm 03.8%	120,770 \pm 01.4%	88,944 \pm 00.7%
		TPC-H-Q2	54,921,837 \pm 0.0%	402,951 \pm 05.7%	221,565 \pm 01.6%	3,461 \pm 04.2%
	1M	TPC-H-Q1	†	863,338 \pm 02.4%	490,846 \pm 00.8%	335,696 \pm 00.9%
		TPC-H-Q2	†	752,133 \pm 04.0%	886,364 \pm 00.1%	12,572 \pm 00.6%
	original	TPC-H-Q1	†	29,345,373 \pm 00.0%	18,630,581 \pm 00.0%	9,371,830 \pm 00.6%
		TPC-H-Q2	†	24,839,845 \pm 00.0%	31,970,932 \pm 00.0%	440,435 \pm 00.8%
SFData	100k	SF-Q1	19,745 \pm 0.1%	10,814 \pm 05.6%	381 \pm 03.7%	374 \pm 04.5%
		SF-Q2	1,395,888 \pm 4.3%	107,340 \pm 01.5%	6,044 \pm 00.3%	9,228 \pm 07.2%
		SF-Q3	1,374,122 \pm 4.3%	101,629 \pm 03.5%	4,710 \pm 03.7%	6,861 \pm 03.3%
	500k	SF-Q1	509,051 \pm 0.1%	24,232 \pm 11.9%	8,343 \pm 02.3%	4,962 \pm 03.7%
		SF-Q2	34,557,816 \pm 0.7%	263,941 \pm 03.5%	149,412 \pm 00.0%	124,786 \pm 05.4%
		SF-Q3	32,563,594 \pm 0.0%	257,551 \pm 01.5%	110,397 \pm 03.1%	99,494 \pm 04.0%
	original	SF-Q1	1,855,169 \pm 0.1%	67,183 \pm 10.7%	65,422 \pm 21.6%	16,849 \pm 01.5%
		SF-Q2	†	878,780 \pm 02.8%	554,790 \pm 01.1%	230,320 \pm 65.6%
		SF-Q3	†	836,345 \pm 00.8%	413,278 \pm 00.4%	316,181 \pm 06.1%
Flight	100k	Flight-Q1	1,191,327 \pm 1.0%	120,986 \pm 57.6%	73,647 \pm 00.4%	280 \pm 48.9%
		Flight-Q2	1,304,863 \pm 2.4%	132,064 \pm 59.9%	73,395 \pm 00.2%	4,844 \pm 03.3%
		Flight-Q3	1,088,810 \pm 2.0%	152,851 \pm 31.4%	36,106 \pm 00.7%	675 \pm 03.1%
		Flight-Q4	1,327,722 \pm 2.5%	155,010 \pm 35.7%	74,711 \pm 00.2%	226 \pm 21.7%
	500k	Flight-Q1	29,650,984 \pm 0.4%	1,043,191 \pm 04.2%	126,333 \pm 02.8%	1,329 \pm 07.0%
		Flight-Q2	16,834,932 \pm 0.6%	1,231,222 \pm 04.1%	129,090 \pm 03.0%	91,106 \pm 02.0%
		Flight-Q3	17,578,355 \pm 0.7%	301,850 \pm 04.7%	72,648 \pm 03.3%	11,569 \pm 01.8%
		Flight-Q4	16,777,141 \pm 0.6%	1,041,470 \pm 15.6%	128,522 \pm 02.9%	2,000 \pm 05.0%
	1M	Flight-Q1	†	800,759 \pm 03.5%	503,496 \pm 00.6%	2,318 \pm 05.6%
		Flight-Q2	67,258,984 \pm 0.0%	925,637 \pm 02.1%	516,183 \pm 00.6%	347,167 \pm 01.1%
		Flight-Q3	74,540,859 \pm 0.0%	870,808 \pm 05.6%	295,086 \pm 01.2%	50,471 \pm 01.1%
		Flight-Q4	66,986,894 \pm 0.0%	855,871 \pm 03.7%	514,542 \pm 00.7%	8,802 \pm 10.3%
	original	Flight-Q1	†	38,000,013 \pm 02.8%	27,037,084 \pm 00.0%	224,395 \pm 04.6%
		Flight-Q2	†	42,649,266 \pm 00.0%	27,662,430 \pm 00.0%	18,298,443 \pm 00.6%
		Flight-Q3	†	40,199,029 \pm 00.0%	15,917,689 \pm 00.0%	2,556,697 \pm 00.8%
		Flight-Q4	†	39,766,269 \pm 00.0%	28,007,283 \pm 00.0%	433,267 \pm 04.5%
Cloud	100k	Cloud-Q1	1,162,777 \pm 0.0%	173,795 \pm 32.0%	65,921 \pm 02.3%	2,634 \pm 10.9%
		Cloud-Q2	1,142,804 \pm 0.4%	223,127 \pm 27.5%	67,101 \pm 01.6%	3,987 \pm 02.3%
		Cloud-Q3	1,178,760 \pm 0.0%	154,900 \pm 39.3%	73,966 \pm 24.5%	2,824 \pm 03.4%
	500k	Cloud-Q1	17,881,648 \pm 0.8%	365,068 \pm 59.0%	115,802 \pm 03.3%	48,062 \pm 01.2%
		Cloud-Q2	20,666,685 \pm 0.1%	356,302 \pm 11.0%	125,579 \pm 03.1%	57,959 \pm 02.8%
		Cloud-Q3	24,579,790 \pm 0.3%	329,993 \pm 06.0%	200,597 \pm 31.5%	52,287 \pm 01.3%
	1M	Cloud-Q1	73,539,017 \pm 0.0%	940,311 \pm 02.3%	460,299 \pm 00.8%	177,332 \pm 00.2%
		Cloud-Q2	84,057,122 \pm 0.0%	1,115,216 \pm 09.2%	497,687 \pm 00.5%	224,698 \pm 01.0%
		Cloud-Q3	†	947,624 \pm 03.7%	897,059 \pm 35.2%	192,318 \pm 00.2%
	5M	Cloud-Q1	†	19,298,715 \pm 00.0%	12,129,946 \pm 00.1%	3,985,490 \pm 00.3%
		Cloud-Q2	†	23,178,835 \pm 00.0%	11,521,068 \pm 00.0%	5,519,018 \pm 01.1%
		Cloud-Q3	†	19,259,093 \pm 00.0%	11,058,125 \pm 04.7%	4,407,499 \pm 00.1%

†: Timeout after 24 hours

*: 2 instead of 20 hyper-threads

◇: 1 instead of 11 nodes

Tab. 3: Average query execution times in ms (and maximum measurement deviations) over five measurements for different subsets of all datasets; on average, the measurement deviations are 0.8% for PostgreSQL, 11.4% for SparkSQL, 3.8% for Redshift and 4.0% for A²DB (if we exclude measurements with sub-second duration). The best execution times are highlighted.

minimal cluster configuration has one leader and one follower node; the largest tested cluster has one leader and eleven follower nodes. We execute query *TPC-H-Q1* (empty result) on *TPC-H 500k* and query *TPC-H-Q2* (relatively large result) on *TPC-H 1M*. Figure 8 plots the query execution times in milliseconds and a reference line for *ideal*, i. e., linear scalability. The measurements for both queries show that the proposed theta-join processing scales linearly with the number of follower nodes; for this reason, we conclude that A²DB’s workload distribution strategy works well. Furthermore, the higher communication costs for larger cluster setups have no major impact on the overall performance, which underlines our observation that A²DB’s theta-join processing is CPU bound.

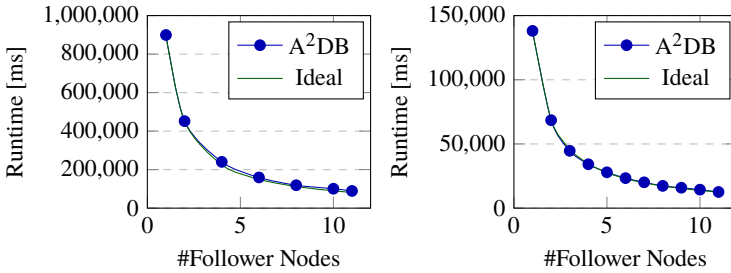


Fig. 8: *TPC-H-Q1* on *TPC-H 500k* and *TPC-H-Q2* on *TPC-H 1M* with varying cluster sizes.

6.4 Work Stealing

As followers sometimes finish their work earlier than others, A²DB implements *work assisting* (WA) and *work stealing* (WS) to keep all nodes well utilized. The next experiment evaluates the effectiveness of the two strategies by comparing the query times with these optimizations to the ones without them. The measurements in Table 4 show that the benefit of balancing workload at the end of the join processing is 6–10% query time reduction. Hence, both strategies effectively accelerate the join processing; the lively redistribution of work between the nodes at the end succeeded to keep all nodes busy.

Recordset	Dataset	Query	Without WA/WS	With WA/WS	Difference
TPC-H	100k	Q1	95,030	88,944	- 6,4%
		Q2	3,873	3,461	- 10,6%
	1M	Q1	373,410	335,696	- 10,1%
		Q2	13,596	12,572	- 7,5%

Tab. 4: Query runtimes on TPC-H with and without *work assisting* and *work stealing*.

6.5 Context-Specific Attribute Provisioning

To reduce the network overhead when fetching remote data, A²DB applies *context-specific attribute provisioning* to send only required values. We now evaluate the effectiveness of

this strategy by measuring the number of attribute values that are transferred and the number of non-relevant attribute values that are filtered. Figure 9 visualizes these numbers for the TPC-H and Flight queries. The measurements show that with context-specific attribute provisioning, we save about 0% (TPC-H-Q1) to 27% (Flight-Q1) values on network traffic. Hence, the savings are dataset-specific, but can be quite substantial. Although not all queries profit from the filtering, most queries in our workload filter at least 10% values in this way.

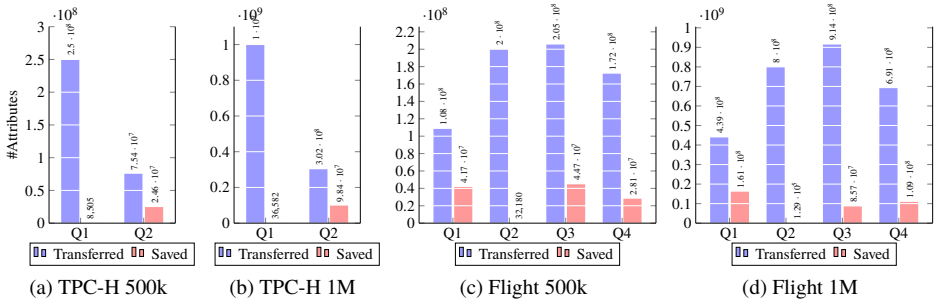


Fig. 9: Amount of transferred (blue) and saved (red) attributes values for the TPC-H and Flight queries with the *context-specific attribute provisioning*.

7 Summary

In this paper, we proposed a novel theta-join algorithm that accelerates the processing of selective theta-join queries via predicate-based candidate pruning and reactive workload distribution. Our experiments show that with these (and probably also further) optimizations, theta-joins can be processed orders of magnitude faster than state-of-the-art join strategies in modern data processing engines. In this way, we motivate a more careful optimization of theta-joins beyond naive nested-loop joins in modern database management systems. The selectivity-based predicate selection approach and the strategy-driven join technique can also be used to extend existing theta-join algorithms, such as IE-Join [Kh15] or 1-Bucket-Theta [Ok11], so that they can handle arbitrary many join predicates as well. Although such combinations could lead to further improvements, their construction and evaluation is not in the scope of this paper and we have to leave them to future work.

References

- [AGN15] Abedjan, Z.; Golab, L.; Naumann, F.: Profiling Relational Data: A Survey. The VLDB Journal 24/4, 2015.
- [Ar15] Armbrust, M.; Xin, R. S.; Lian, C.; Huai, Y.; Liu, D.; Bradley, J. K.; Meng, X.; Kaftan, T.; Franklin, M. J.; Ghodsi, A.; Zaharia, M.: Spark SQL: Relational Data Processing in Spark. In: Proceedings of the International Conference on Management of Data (SIGMOD). 2015.

- [BC81] Bernstein, P. A.; Chiu, D.-M. W.: Using Semi-Joins to Solve Relational Queries. *Journal of the ACM* 28/1, 1981.
- [Be18] Bernstein, P. A.: Actor-Oriented Database Systems. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 2018.
- [Bo07] Bohannon, P.; Fan, W.; Geerts, F.; Jia, X.; Kementsietsidis, A.: Conditional Functional Dependencies for Data Cleaning. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 2007.
- [Co17] Cong, G.; Fan, W.; Geerts, F.; Jia, X.; Ma, S.: Improving Data Quality: Consistency and Accuracy. In: *Proceedings of the VLDB Endowment*. 2017.
- [Co79] Codd, E. F.: Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems (TODS)* 4/4, 1979.
- [ESW78] Epstein, R.; Stonebraker, M.; Wong, E.: Distributed Query Processing in a Relational Data Base System. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1978.
- [Go75] Gotlieb, L. R.: Computing Joins of Relations. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1975.
- [Gu15] Gupta, A.; Agarwal, D.; Tan, D.; Kulesza, J.; Pathak, R.; Stefani, S.; Srinivasan, V.: Amazon Redshift and the Case for Simpler Data Warehouses. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2015.
- [Kh15] Khayyat, Z.; Lucia, W.; Singh, M.; Ouzzani, M.; Papotti, P.; Quiané-Ruiz, J.-A.; Tang, N.; Kalnis, P.: Lightning Fast and Space Efficient Inequality Joins. In: *Proceedings of the VLDB Endowment*. 2015.
- [KNG18] Koumarelas, I.; Naskos, A.; Gounaris, A.: Flexible partitioning for selective binary theta-joins in a massively parallel setting. *Distributed and Parallel Databases* 36/2, 2018.
- [ME92] Mishra, P.; Eich, M. H.: Join Processing in Relational Databases. *ACM Computing Surveys* 24/1, 1992.
- [Ok11] Okcan, A.: Processing Theta-Joins Using MapReduce. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2011.
- [SSP19] Schmidl, S.; Schneider, F.; Papenbrock, T.: An Actor Database System for Akka. In: *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW)*. 2019.
- [VG84] Valduriez, P.; Gardarin, G.: Join and Semijoin Algorithms for a Multiprocessor Database Machine. *ACM Transactions on Database Systems (TODS)* 9/1, 1984.
- [ZCW12] Zhang, X.; Chen, L.; Wang, M.: Efficient Multi-Way Theta-Join Processing Using MapReduce. In: *Proceedings of the VLDB Endowment*. 2012.

Precise, Compact, and Fast Data Access Counters for Automated Physical Database Design

Michael Brendle,¹ Nick Weber^{2*}, Mahammad Valiyev^{3*}, Norman May⁴, Robert Schulze⁴,
Alexander Böhm⁴, Guido Moerkotte⁵, Michael Grossniklaus⁶

Abstract: Today's database management systems offer numerous tuning knobs that allow an adaptation of database system behavior to specific customer needs, e. g., maximal throughput or minimal memory consumption. Because manual tuning by database experts is complicated and expensive, academia and industry devised tools that automate physical database tuning. The effectiveness of such advisor tools strongly depends on the availability of accurate statistics about the executed database workload. For advisor tools to run online, workload execution statistics must also be collected with low runtime and memory overhead. However, to the best of our knowledge, no approach collects precise, compact, and fast workload execution statistics for a physical database design tool. In this paper, we present data structures that solve the problem of providing workload execution statistics with high precision, low memory consumption, and low runtime overhead. In particular, we show how existing approaches can be combined and for which advisor tools, new data structures need to be designed. We evaluate our data structures in a prototype of a commercial database and show that they outperform previous approaches using real-world and synthetic benchmarks.

1 Introduction

Modern database management systems (DBMS) offer a plethora of tuning knobs to adapt the system behavior to specific customer needs [Ag04; Ra02]. As a result, finding an optimal configuration that meets all requirements (e. g., with respect to throughput or memory consumption) is usually a difficult task performed by experts. Since manual database tuning by experts is expensive or even infeasible in managed database-as-a-service (DBaaS) environments, academia and industry devised tools for automated physical database design [Lu19]: **(1) Index advisors** improve query performance by creating (clustered) indexes on columns frequently referenced in selective query predicates [Ag04; Ko20; Na20]. **(2) Data compression advisors** reduce the table memory consumption, and thus, the amount of data read and processed by physically compacting columns [Da19; Le10]. **(3) Buffer pool size advisors** lower the Total Cost of Ownership (TCO) by setting the buffer

¹ University of Konstanz, P.O. Box 188, 78457 Konstanz, Germany michael.brendle@uni-konstanz.de

² Celonis SE, Theresienstr. 6, 80333 Munich, Germany n.weber@celonis.com

³ Technical University of Munich, Boltzmannstraße 3, 85748 Garching mahammad.valiyev@tum.de

⁴ SAP SE, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany firstname.lastname@sap.com

⁵ University of Mannheim, 68131 Mannheim, Germany moerkotte@uni-mannheim.de

⁶ University of Konstanz, P.O. Box 188, 78457 Konstanz, Germany michael.grossniklaus@uni-konstanz.de

* Work done while at SAP SE

pool size to the working set size such that memory costs are minimized without impairing performance [Da16; St06]. Finally, **(4) table partitioning advisors** enable partition pruning, an effective method of reducing the amount of data to be read [ABI19; Ag04; Cu10; Ra02; Se16]. Furthermore, separating frequently accessed (hot) and rarely accessed (cold) data into disjoint partitions can increase the buffer pool hit ratio.

All aforementioned physical database design tools require an objective function, e. g., the workload performance or memory footprint, while respecting given constraints, e. g., a memory budget or maximum workload execution time. To do this, advisor tools consider a set of potential new physical layout alternatives (e. g., by enumeration). For each alternative, the advisor calculates a change in the objective function based on the data, the workload, and the current physical layout. Accurate statistics about the executed workload are of particular importance for the effectiveness of many advisors. For example, index advisors rely on precise knowledge of query predicate selectivities, data compression advisors depend on understanding how much data is sequentially read (e. g., scans) or randomly accessed (e. g., index join), buffer pool size advisors are based on page access statistics, whereas table partitioning advisors build upon row- or value-level access statistics.

Obviously, there is a trade-off between the accuracy of workload execution statistics and their runtime and memory overhead. Ideally, workload execution statistics are collected with low overhead, such that advisor tools can be executed online to adapt to dynamically changing workloads. However, in practice, workload execution statistics are either gathered offline, e. g., by executing a representative sample of the workload on a separate node [Ag04; Cu10; Ra02], or collected with low precision, e. g., by tracking access frequencies at page granularity instead of per row and attribute, combined with sampling [FKN12; Hu19; No20]. As a result, to the best of our knowledge, no approach collects precise, compact, and fast workload execution statistics for an advisor tool.

In this work, we formalize, analyze, and solve the problem of providing workload execution statistics with high precision, low memory consumption, and low runtime overhead as input to automated physical database design tools. Our contributions are as follows:

- we demonstrate and discuss four practical use cases of automated physical database design advice that require workload execution statistics as input (Section 2);
- we define the workload execution statistics that need to be collected, and we subsequently formalize the problem (Section 3);
- we discuss and classify related work with respect to their precision, space efficiency, and runtime overhead (Section 4);
- we present data structures for collecting precise, compact, and fast workload execution statistics (Section 5); and
- we implement our data structures prototypically in SAP HANA and show for each use case that workload execution statistics are provided with high precision and low memory and runtime overhead using real-world and synthetic benchmarks (Section 6).

2 Use Cases of Physical Database Design Advice

This section introduces four use cases of automated physical database design advice in column stores that require workload execution statistics $FStat$ about a workload W . For now, it suffices to think of W as a set of SQL statements and $FStat$ as statistics about W collected during the execution of W .

We argue that automated physical database design tools can be categorized according to their objective function, aiming either for maximum performance or minimum memory footprint. Besides that, advisor tools need to fulfill given constraints, e. g., a memory budget or a maximum workload execution time. In Section 2.1, we introduce an index advisor and a data compression advisor that focus on in-memory performance, i. e., speeding up query response times of given workloads. Section 2.2 presents a buffer pool size advisor and a table partitioning advisor that optimize for memory footprint.

In the following, \mathcal{R} denotes a set of n relations, and $\mathcal{A}(R_i)$ is the set of m_i attributes of relation $R_i \in \mathcal{R}$. Further, $D(A_{i,j}) = \{v_{i,j,1}, \dots, v_{i,j,k}, \dots, v_{i,j,d_{i,j}}\}$ refers to the active domain of attribute $A_{i,j} \in \mathcal{A}(R_i)$ with $v_{i,j,1} < \dots < v_{i,j,k} < \dots < v_{i,j,d_{i,j}}$, where $d_{i,j}$ is the number of distinct values in $A_{i,j}$. Finally, $R_i[\text{rid}_i].A_{i,j} \in D(A_{i,j})$ is the value of the row with row id $\text{rid}_i \in [1, |R_i|]$ of attribute $A_{i,j} \in \mathcal{A}(R_i)$, where $|R_i|$ is the cardinality of $R_i \in \mathcal{R}$.

2.1 Automated Physical Database Design for Maximizing Performance

Creating a (clustered) index on a column improves the performance if the workload includes selective filter predicates. Traversing the index is then faster than performing a full column scan. Besides that, we assume that a memory budget is given to create indexes only on those attributes where they yield the largest benefit [Ag04; Ko20; Ra02].

Use Case 1 (Index Advisor) *Let $\mathbb{A}_{i,s} \in \wp(\mathcal{A}(R_i))$ be a set of attributes from the power set of all attributes that is uniquely identified by $s \in [1, |\wp(\mathcal{A}(R_i))|]$, $I_{i,s}$ a single-/multi-column index defined over $\mathbb{A}_{i,s}$, and \mathbb{I} the set of all possible indexes over all relations. An index advisor proposes an index configuration $IC \subseteq \mathbb{I}$ such that the estimated execution time \widehat{E} of a workload W based on workload execution statistics $FStat$ is minimized while the estimated additional memory consumption \widehat{M} of the indexes adheres to a given memory budget MB :*

$$\arg \min_{IC \subseteq \mathbb{I}} \widehat{E}(IC, W, FStat) \quad \text{subject to } \widehat{M}(IC) \leq MB.$$

Applying compression to a column may reduce its size, and thus, the amount of data processed by sequential scans. In contrast, compression may deteriorate the time to dereference individual row ids (e. g., during projections) since the decompression of individual rows or blocks may incur multiple random memory accesses, depending on the

compression technique. In practice, robust performance is often preferred, and a column would only be compressed if the speed of critical SQL statements does not decline compared to an uncompressed column [Da19; Le10]).

Use Case 2 (Data Compression Advisor) Let $\mathbb{C}_{i,j}$ be a set of compressed and uncompressed storage layouts for an attribute $A_{i,j} \in R_i$, $C_{i,j}^u \in \mathbb{C}_{i,j}$ be the uncompressed storage layout, and $W_{crit} \subseteq W$ be the subset of (business) critical SQL statements in the workload, defined by the user. A data compression advisor proposes for each attribute $A_{i,j} \in \mathcal{A}(R_i)$ of each relation $R_i \in \mathcal{R}$ a physical storage layout $C_{i,j} \in \mathbb{C}_{i,j}$ such that the estimated execution time \widehat{E} of a workload W based on workload execution statistics $FStat$ is minimized, while for each critical SQL statement $q \in W_{crit}$, the estimated execution time \widehat{E} does not exceed the estimated execution time \widehat{E} without compression:

$$\begin{aligned} & \arg \min_{\forall R_i \in \mathcal{R} \forall A_{i,j} \in \mathcal{A}(R_i): C_{i,j} \in \mathbb{C}_{i,j}} \widehat{E}(\{C_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq m_i\}, W, FStat) \\ & \text{subject to} \quad \forall q \in W_{crit} : \widehat{E}(\{C_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq m_i\}, q, FStat) \\ & \quad \leq \widehat{E}(\{C_{i,j}^u \mid 1 \leq i \leq n, 1 \leq j \leq m_i\}, q, FStat). \end{aligned}$$

2.2 Automated Physical Database Design for Memory Footprint Reduction

A buffer pool size advisor aims for a minimal buffer pool size such that a performance constraint, e. g., a maximum workload execution time, is still fulfilled. To do this, a buffer pool size advisor needs to identify the workload's working set and configure the buffer pool size so that all hot pages can still be held in DRAM.

Use Case 3 (Buffer Pool Size Advisor) A buffer pool size advisor proposes a minimal buffer pool size $B \in \mathbb{N}$ such that the estimated execution time \widehat{E} of a workload W based on workload execution statistics $FStat$ does not violate a given threshold SLA :

$$\begin{aligned} & \arg \min_{B \in \mathbb{N}} B \\ & \text{subject to} \quad \widehat{E}(B, W, FStat) \leq SLA. \end{aligned}$$

A buffer pool is a simple and practical approach to retain data's hot working set in DRAM. Its most significant drawback is that mixing hot and cold data within the same page pollutes the buffer cache and works against its effectiveness. Table range partitioning separates hot and cold data into disjoint range partitions, and hence, improves the buffer pool hit ratio.

Use Case 4 (Table Partitioning Advisor) Let \mathbb{S}_i be a set of range partitioning specifications for a relation $R_i \in \mathcal{R}$. A table partitioning advisor proposes a buffer pool size $B \in \mathbb{N}$, and for each relation $R_i \in \mathcal{R}$ a range-partitioning $S_i \in \mathbb{S}_i$ such that the buffer pool size B is minimized, while the estimated execution time \widehat{E} of workload W with workload execution statistics $FStat$ does not violate a maximum workload execution time SLA .

$$\begin{aligned} & \arg \min_{B \in \mathbb{N}, R_i \in \mathcal{R}: S_i \in \mathbb{S}_i} B \\ & \text{subject to} \quad \widehat{E}(\{S_i \mid 1 \leq i \leq n\}, B, W, FStat) \leq SLA. \end{aligned}$$

3 Problem Statement

We now formalize the problem of providing workload execution statistics $FStat$ with high precision, low memory consumption, and low runtime overhead as input to automated physical database design tools. We start this section by defining $FStat$ for a workload W and show exemplary $FStat$ after executing JCC-H Query 3 [BAK18].

Definition 1 (Workload Execution Statistics) We define a workload W as a multiset of SQL statements⁷ and $T(q)$ as the physical execution plan of a SQL statement $q \in W$. For a workload W , we define workload execution statistics $FStat$:

- F1 (Index Advisor):* For each executed SQL statement $q \in W$, $FStat$ stores for each selection $\sigma_p(R_i)$ on a base relation $R_i \in \mathcal{R}$ in the physical execution plan $T(q)$ that consists of an index-SARGable predicate p , a tuple $(|\sigma_p(R_i)|, \mathcal{F}(p))$, where $|\sigma_p(R_i)|$ is the output cardinality of $\sigma_p(R_i)$ and $\mathcal{F}(p)$ are the free attributes contained in p .
- F2 (Data Compression Advisor):* For each executed SQL statement $q \in W$, $FStat$ stores for each attribute $A_{i,j} \in R_i$ a pair $(s_{i,j}, r_{i,j})$, where $s_{i,j}$ is the number of rows in $A_{i,j}$ that were sequentially accessed by q (e. g., by a selection $\sigma_p(R_i) \in T(q)$, where p contains $A_{i,j}$), and $r_{i,j}$ is the number of rows that were randomly accessed in $A_{i,j}$ by q (e. g., by a projection $\Pi_{A_{i,j}} \in T(q)$).
- F3 (Buffer Pool Size Advisor):* For each executed SQL statement $q \in W$, $FStat$ stores the access frequency $f_{P_{i,j,u}}$ to each page $P_{i,j,u} \in \mathbb{P}_{i,j}, u \in [1, |\mathbb{P}_{i,j}|]$ (i. e., $P_{i,j,u}$ stores for a set of rows the values $R_i[\text{rid}_i].A_{i,j}$), where $\mathbb{P}_{i,j}$ is the set of all pages of $A_{i,j} \in R_i$.
- F4 (Table Partitioning Advisor):* For each executed SQL statement $q \in W$, $FStat$ stores the access frequency $f_{v_{i,j,k}}$ for each value $v_{i,j,k} \in D(A_{i,j})$, where $f_{v_{i,j,k}}$ is the sum of
- the number of sequential reads of $A_{i,j}$ by q such that $\exists R_i[\text{rid}_i].A_{i,j} = v_{i,j,k}, \text{rid}_i \in [1, |R_i|]$ that is part of the matching rows (e. g., by a selection $\sigma_p(e) \in T(q)$ where p references $A_{i,j}$ and $v_{i,j,k}$ satisfy p)⁸, and
 - the number of random reads of rows in $A_{i,j}$ by q such that $R_i[\text{rid}_i].A_{i,j} = v_{i,j,k}, \forall \text{rid}_i \in [1, |R_i|]$ (e. g., by a projection $\Pi_{A_{i,j}} \in T(q)$).

We execute JCC-H Q3 [BAK18] to demonstrate $FStat$.

Figure 1 shows the optimal query execution plan, identified by SAP HANAs query optimizer [MBL17].

$ \sigma_p(R_i) $	$\mathcal{F}(p)$
3,774,696	{ O_ORDERDATE }
299,496	{ C_MKTSEGMENT }

Table 1 shows $FStat$ *F1* for an index advisor.

Since the most selective predicate is applied to C_MKTSEGMENT, an index advisor might propose an index on this attribute. Depending on the memory budget, the advisor might also recommend an index on O_ORDERDATE. The selection on L_SHIPDATE is not recorded since it is not performed on a base relation in the plan.

Tab. 1: Collected statistics $FStat$ *F1* for selections $\sigma_p(R_i)$ of JCC-H Q3.

⁷ We consider multisets of SQL statements to account for realistic workloads with repeated queries.

⁸ We record only accesses to rows that match the predicate since we assume that a range partition generated for a value $v_{i,j,k}$ is pruned if the value does not satisfy the predicate.

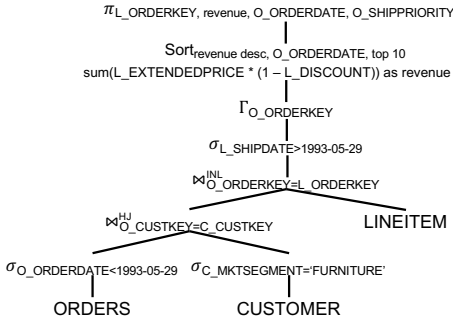


Fig. 1: Optimal query execution plan for JCC-H Q3, identified by SAP HANAs query optimizer.

$A_{i,j}$	$s_{i,j}$	$r_{i,j}$
C_CUSTKEY	0	299,496
C_MKTSEGMENT	1,500,000	0
O_ORDERKEY	0	1,015,311
O_CUSTKEY	0	3,774,696
O_ORDERDATE	15,000,000	377,432
O_SHIPPRIORITY	0	10
L_ORDERKEY	0	3,045,935
L_DISCOUNT	0	1,074,616
L_EXTENDEDPRICE	0	1,074,616
L_SHIPDATE	0	3,045,935

Tab. 2: Collected statistics $FStat F2$ about the number of rows that were sequentially ($s_{i,j}$) and randomly ($r_{i,j}$) read for each $A_{i,j}$.

Table 2 shows $FStat F2$ for a data compression advisor. Since C_MKTSEGMENT exposes only sequential but no random reads, a data compression advisor might suggest compression. A data compression advisor might also propose compression of O_ORDERDATE since the amount of data processed by sequential scans is reduced. However, random accesses would slow down the time of dereferencing individual row ids due to compression. Therefore, the data compression advisor needs to consider the trade-off between the gain of speeding up sequential reads and the loss of slowing down random accesses.

Figure 2 shows for each 256KB page $P_{i,j,u}$ (x-axis) of L_EXTENDEDPRICE the access frequency $f_{P_{i,j,u}}$ (y-axis), i. e., $FStat F3$. Due to dictionary compression in SAP HANA [MBL17], pages contain either value-id array chunks (600 pages) or dictionary data (40 pages). Since only $\approx 75\%$ of the value-id array pages are accessed, a buffer pool size advisor might propose reducing the buffer pool size such that all hot pages can still be held in DRAM.

Figure 3 shows for each value $v_{i,j,k}$ of the active domain of O_ORDERDATE (x-axis) the access frequency $f_{v_{i,j,k}}$ (y-axis), i. e., $FStat F4$. A table partitioning advisor might propose a (hot) range-partition for data items with O_ORDERDATE between 1993-01-29 and 1993-05-28 since only those values are accessed frequently. In contrast, data items with O_ORDERDATE larger than 1993-05-28 have an access frequency of 0 and a corresponding (cold) table partition will be pruned by the predicate on O_ORDERDATE.

Problem 1 *The problem we consider is to provide workload execution statistics $FStat$, which are precise (i. e., as accurate as possible), compact (i. e., the memory footprint compared to the data set size should be as small as possible), and fast (i. e., the runtime overhead during workload execution should be as low as possible).*

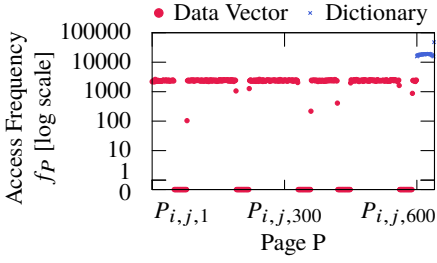


Fig. 2: Collected statistics $FStat\ F3$ about the access frequency $f_{P_{i,j,u}}$ of each page $P_{i,j,u}$ of $L_EXTENDEDPRICE$ for JCC-H Q3.

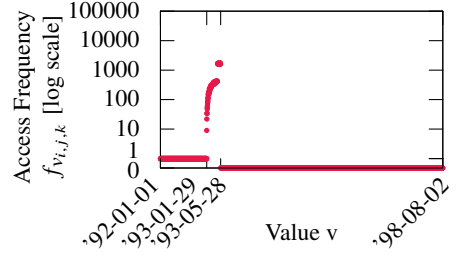


Fig. 3: Collected statistics $FStat\ F4$ about the access frequency $f_{v_{i,j,k}}$ of each value $v_{i,j,k} \in D(O_ORDERDATE)$ for JCC-H Q3.

4 Related Work

This section discusses and classifies related approaches of collecting workload execution statistics $FStat$ with respect to the precision for use cases F1 to F4, space efficiency, and runtime overhead. The considered approaches are summarized in Table 3.

The first type of workload execution statistics are **row-level data access counters**. Project Siberia [LLS13] analyzes log samples to estimate the access frequency of rows, and SAP ASE [Gu18] caches runtime access patterns of rows. In row stores, this approach yields precise access frequencies of pages (F3). To also track access frequencies of active domain values precisely (F4), separate counters per domain value and attribute are needed, which results in high memory consumption and runtime overhead. Furthermore, with row-level counting, it is unable to deduce the output cardinality of selections (F1). Finally, the total number of rows that were accessed sequentially or randomly can only be tracked if separate counters of each access type exist (F2).

Another class of workload execution statistics are **graphs**. In Schism [Cu10] and Clay [Se16], each row is represented as a node, and edges connect rows if accessed within the same transaction. The weight of an edge denotes the number of transactions that accessed both rows. Graphs are as precise as row-level data access counters. However, the memory and runtime overhead depends on the workload. If transactions touch only a few rows, an adjacency list results in low memory and runtime overhead. In contrast, if transactions touch many rows, both an adjacency list or a matrix result in high memory and runtime overhead.

To further improve the memory and runtime overhead, **block-level data access counters** were proposed. For example, X-Engine [Hu19] leverages access frequencies at extent level collected during workload execution, and HyPer [FKN12] uses for each virtual memory page flags of the CPU’s MMU to identify cold pages. Block-level data access counters provide precise access frequencies of pages (F3). The tracking accuracy for accesses to the active domain (F4) depends heavily on the workload and falls short in the presence of heavy

Approach for collecting workload execution statistics	Precise $FStat$				Compact	Fast
	$F1$	$F2$	$F3$	$F4$		
Row-level data access counters [Gu18; LLS13]	✗	✓	✓	✓	✗	✗
Graph representation [Cu10; Se16]	✗	✓	✓	✓	●	●
Block-level data access counters [FKN12; Hu19]	✗	✓	✓	●	✓	●
SQL statements + What-if API [Ag04; Ra02]	●	●	●	●	✓	✗
Memory access tracing [No20]	✗	✗	✓	✓	✗	✗
Our approach	✓	✓	✓	✓	✓	✓

Tab. 3: Comparison between different approaches for collecting workload execution statistics $FStat$ as input to advisor tools with respect to their precision, space efficiency, and runtime overhead.

hitters. The total number of rows sequentially or randomly accessed is available if separate counters for each access type are maintained (F2). The access granularity cannot be tracked as row-level access counters (F1). While block-level access counters are compact, their runtime overhead depends on the workload. In the worst-case, all counters of all blocks accessed need to be incremented (e. g., during a full column scan).

A traditional approach of collecting workload execution statistics is to feed the workload’s **SQL statements** into offline physical design advisors, which rely on the query optimizer’s **what-if API** [Ag04; Ra02]. While the collected SQL statements are compact, the most significant drawback is that physical accesses to the data are not tracked. Thus, the approach fails to provide accurate statistics as it relies on estimates.

Instead of collecting workload execution statistics inside the database, **memory access tracing** [No20] uses the PEBS mechanism of Intel processors to trace memory accesses, which are mapped to the data to determine precise access frequencies of pages (F3) and values of the active attribute domain (F4). While only single memory accesses are traced, the access granularity (F1) and access type (F2) cannot be identified. Since memory traces are logged and analyzed offline, the memory and runtime overhead is high.

In sum, no approach collects precise, compact, and fast workload execution statistics $FStat$ for a physical database design tool. In the next section, we show how existing approaches can be combined and for which advisor tools new data structures need to be designed.

5 Data Access Counters

We begin describing our approach by explaining how precise, compact, and fast workload execution statistics for an index advisor can be collected (Section 5.1). Afterwards, we present data structures for a data compression advisor (Section 5.2), a buffer pool size advisor (Section 5.3), and a table partitioning advisor (Section 5.4).

5.1 Use Case 1: Index Advisor

The most popular approaches of providing workload execution statistics for index advisors (*FStat FI*) consider SQL statements as input to the optimizer's what-if API. As a result, those approaches are limited in their performance due to what-if analysis and rely on the availability of precise cardinality estimates. To address these limitations, we track the actual output cardinalities of selections $\sigma_p(R_i)$ at query execution time. Since tracking the exact output cardinalities $|\sigma_p(R_i)|$ of all selections would consume too much memory, we introduce a threshold parameter $\phi \in (0, 1]$ to capture only selections with an output cardinality less than $\phi \cdot |R_i|$ since only selective predicates benefit from indexes [KAI17]. To reduce the memory overhead further, we group the actual output cardinalities into intervals $[b^r, b^{r+1})$, $b \in \mathbb{R}_{>0}$, $0 \leq r \leq \lceil \log_b(\phi \cdot |R_i|) \rceil$ and instead only count the number of selections per interval. The estimated output cardinality for selections that are recorded to the interval $[b^r, b^{r+1})$ is $\sqrt{b^r \cdot b^{r+1}}$. Hence, we determine an error (i. e., the ratio between the actual and recorded output cardinality) of \sqrt{b} for arbitrary complex predicates. In our experiments in Section 6, we set the interval base parameter b to 2, such that the actual and recorded output cardinalities differ at most by a factor of $\sqrt{2}$.

Since an index advisor may recommend multi-column indexes, we would need one set of intervals (i. e., $[b^r, b^{r+1})$, $b \in \mathbb{R}_{>0}$, $0 \leq r \leq \lceil \log_b(\phi \cdot |R_i|) \rceil$) per combination of free attributes per relation, i. e., in total, $2^{m_i} - 1$ ($= |\wp(\mathcal{A}(R_i)) \setminus \{\emptyset\}|$) set of intervals. As a result, the memory consumption of our approach using 32-bit counters for a relation R_i with m_i attributes would be $(\lceil \log_b(\phi \cdot |R_i|) \rceil + 1) \cdot (2^{m_i} - 1) \cdot 4$ bytes. To meet the memory requirements, we propose lazy counters, only created if (1) the corresponding combination of free attributes actually occurred in selection predicates and (2) the selectivity of this attribute combination is below ϕ . We argue that this number of attribute combinations is significantly smaller than the number of all attribute combinations. For example, for LINEITEM with scale factor 10 (i. e., 16 attributes and 60,000,000 rows) and $b = 2$, counters for all combinations of free attributes constitute 0.32% of the data set size of LINEITEM in SAP HANA (1.90 GB), while our lazy counters constitute only 0.02% of the data set size.

Section 6 demonstrates that our approach has a high precision as well as a low memory and runtime overhead. We summarize the presented data structure in the following:

Access Counter 1 (Index Advisor)

Physical Accesses: We consider each selection $\sigma_p(R_i)$ consisting of an index-SARGable predicate, and its actual output cardinality $|\sigma_p(R_i)|$, collected during query execution.

Lazy Counters: For a base $b \in \mathbb{R}$ and a set of attributes $\mathbb{A}_{i,s} \in \wp(\mathcal{A}(R_i))$, we create and maintain integer counters $X_{i,s,0}^{idx}, \dots, X_{i,s,r}^{idx}, \dots, X_{i,s,\lceil \log_b(\phi \cdot |R_i|) \rceil}^{idx}$ if there exists a selection $\sigma_p(R_i) \in T(q)$, $q \in W$ such that $\mathbb{A}_{i,s} \subseteq \mathcal{F}(p)$ and $|\sigma_p(R_i)| < \phi \cdot |R_i|$.

Interval Counting: A counter $X_{i,s,r}^{idx}$ is incremented by 1 for a selection $\sigma_p(R_i) \in T(q)$, $q \in W$ if $|\sigma_p(R_i)| > 0$ and $r = \lceil \log_b(|\sigma_p(R_i)|) \rceil$ and $|\sigma_p(R_i)| < \phi \cdot |R_i|$. For $|\sigma_p(R_i)| = 0$, $X_{i,s,0}^{idx}$ is incremented by 1.

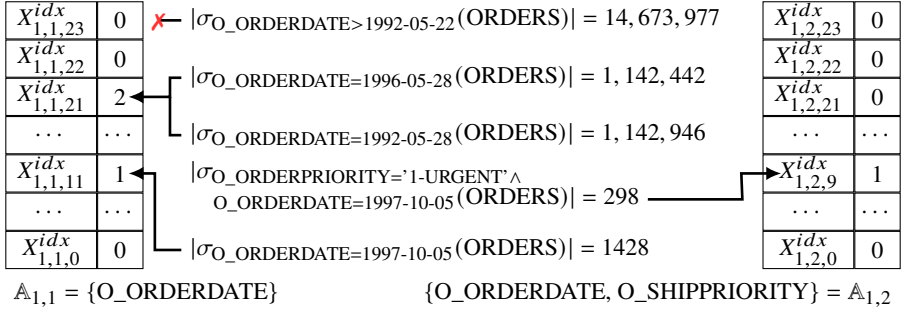


Fig. 4: Illustration of our approach for collecting workload execution statistics for an index advisor.

Figure 4 shows for five selections on *ORDERS* with scale factor 10 (15,000,000 rows) how the access counters with base $b = 2$ are updated. We show the access counters for selection predicates containing attribute *O_ORDERDATE* (left), and selection predicates containing *O_ORDERDATE* and *O_ORDERPRIORITY* (right). The first selection on *O_ORDERDATE* matches 14,673,977 rows, and thus no counter is updated for $\phi = 0.1$. The counter $X_{1,1,21}^{idx}$ is updated twice, by the second ($\lceil \log_2(1,142,442) \rceil = 21$) and the third selection ($\lceil \log_2(1,142,946) \rceil = 21$). The fourth selection updates the counter $X_{1,2,9}^{idx}$ for the attribute set of *O_ORDERDATE* and *O_SHIPPRIORITY* as 298 rows match, and two attributes are referenced in the predicate.

As future work, we plan to collect for a join $e \bowtie_{A_{i,j}=A_{i,j}} R_i$, where e is an expression (e.g., $\sigma_p(R_i)$), the cardinality of expression e (i.e., $|e|$) for attribute $A_{i,j}$ of relation R_i . The reason is that an index on an attribute $A_{i,j}$ may improve the performance if $|e|$ is small. Traversing the index on $A_{i,j}$ is then faster than building a hash table on $A_{i,j}$.

5.2 Use Case 2: Data Compression Advisor

In Section 4, we have shown that existing approaches of collecting workload execution statistics for data compression advisors (*FStat F2*) do not consider the type of access (i.e., sequential vs. random access). We propose to count both the number of rows accessed sequentially and randomly by the workload. Maintaining just two counters per attribute fulfills the space efficiency requirement. Section 6 shows that our approach also achieves a low runtime overhead. Note that besides workload execution statistics, characteristics of the data (e.g., number of distinct values, value distribution, or whether the data is sorted) are also needed to propose an optimal compression layout (Use Case 2) [Da19]. Moreover, these statistics are typically available in databases today with sufficient quality. However, workload execution statistics are essential in estimating the performance benefit, particularly for (business) critical queries. We summarize the presented access counter in the following:

Access Counter 2 (Data Compression Advisor)

Physical Accesses: We consider the physical data accesses during execution of workload W .

Access Type: For each attribute $A_{i,j} \in R_i$, we create and maintain an integer counter $X_{i,j}^s$, which tracks the number of rows sequentially read, and an integer counter $X_{i,j}^r$, which tracks the number of rows randomly accessed.

$A_{i,j}$	$X_{i,j}^s$	$X_{i,j}^r$
C_CUSTKEY	0	299,496
C_MKTSEGMENT	1,500,000	0
O_ORDERKEY	0	1,015,311
O_CUSTKEY	0	3,774,696
O_ORDERDATE	15,000,000	377,432
O_SHIPRIORITY	0	10
L_ORDERKEY	0	3,045,935
L_DISCOUNT	0	1,074,616
L_EXTENDEDPRICE	0	1,074,616
L_SHIPDATE	0	3,045,935

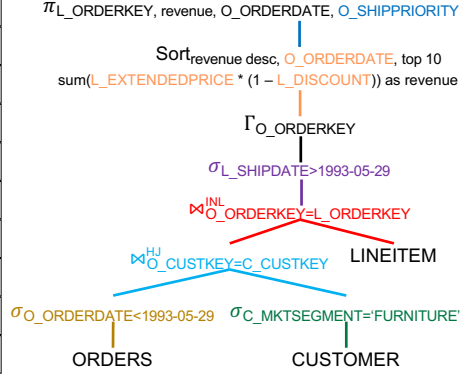


Fig. 5: Illustration of the data structure for collecting $FStat F2$ for a data compression advisor.

Figure 5 shows for JCC-H Q3 how $X_{i,j}^s$ and $X_{i,j}^r$ are updated. Note that these statistics are actual values from the execution with SAP HANA. Data accesses by an operator in the plan and updating the corresponding counter are highlighted using a unique color. The selection on O_ORDERDATE causes 15,000,000 sequential row accesses, while the join between ORDERS and CUSTOMER causes 299,496 random row accesses to C_CUSTKEY and 3,774,696 random accesses to O_CUSTKEY (a customer has on average 10 orders). The projection on O_SHIPRIORITY generates 10 random row accesses due to the top-10 query.

5.3 Use Case 3: Buffer Pool Size Advisor

Block-level data access counters provide precise access frequencies of pages if the block size equals the page size. However, keeping track of accesses that span multiple pages requires updating $|\mathbb{P}_{i,j}|$ -many block counters. Instead of updating for each query the frequencies of all touched pages individually, we propose to update only the respective start and end page counters: If a query accesses the pages $[P_{i,j,v}, P_{i,j,w})$, $P_{i,j,v}, P_{i,j,w} \in \mathbb{P}_{i,j}$, the corresponding counter to page $P_{i,j,v}$ is incremented, while the counter of page $P_{i,j,w+1}$ is decremented since $P_{i,j,w}$ is the last accessed page. This enables counter updates in constant time. Since we decrement the counter of the following page, in total $|\mathbb{P}_{i,j}| + 1$ counters are needed to be able to decrement a counter for accesses to the last page $P_{i,j}, |\mathbb{P}_{i,j}|$. After

statistics collection, the final page access frequencies are derived by calculating the prefix sum of the counters up to the target page. We argue that the statistics are considerably more often updated than read (e. g., after a sampling phase) and that we thus meet the runtime overhead requirements. Furthermore, the memory overhead is low because only a single 64-bit signed integer counter per page is stored. For example, in SAP HANA [Sh19] the memory footprint varies between 0.2% (64 bit/4 KB) and 0.00005% (64 bit/16 MB), depending on the page size. We present the data structure below:

Access Counter 3 (Buffer Pool Size Advisor)

Physical Accesses: We consider the physical data accesses by the workload W .

Start/End Block Counting: For each attribute $A_{i,j} \in R_i$, we create and maintain integer counters $X_{i,j,1}^P, \dots, X_{i,j,v}^P, \dots, X_{i,j,(\lceil \mathbb{P}_{i,j} \rceil + 1)}^P$. For physical accesses to pages in the range $[P_{i,j,v}, P_{i,j,w}]$, $P_{i,j,v}, P_{i,j,w} \in \mathbb{P}_{i,j}$, counter $X_{i,j,v}^P$ is incremented by 1, and counter $X_{i,j,(w+1)}^P$ is decremented by 1. The access frequency $f_{P_{i,j,u}}$ for page $P_{i,j,u}$ is defined as $f_{P_{i,j,u}} = \sum_{v=1}^u X_{i,j,v}^P$.

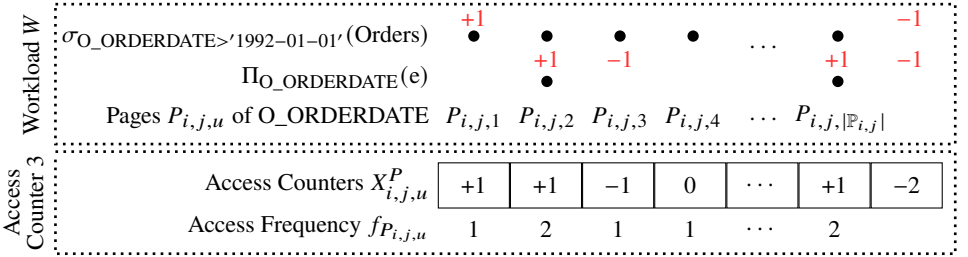


Fig. 6: Illustration of the data structure for collection $FStat F3$ for a buffer pool size advisor.

Figure 6 shows for a selection and a projection on O_ORDERDATE how the page accesses are counted. The selection changes counter $X_{i,j,1}^P$ by +1 and counter $X_{i,j,(\lceil \mathbb{P}_{i,j} \rceil + 1)}^P$ by -1, while the projection increments only the counter of the accessed page by +1 and decrements the counter of the following page by -1. Note that for accesses to the last page, the counter $X_{i,j,(\lceil \mathbb{P}_{i,j} \rceil + 1)}^P$ is decremented. We compute the prefix sum of the counters up to the target page to obtain the access frequencies of individual pages, e. g., page $P_{i,j,2}$ has an access frequency of 2 ($= X_{i,j,1}^P + X_{i,j,2}^P$).

5.4 Use Case 4: Table Partitioning Advisor

A naïve approach of tracking the access frequencies of values in the active attribute domain ($FStat F4$) is to group values into value ranges and to increment a value range counter by one whenever a value or sub-range of the value range is read. With the counter representing

the access frequency of each value in the range, frequencies are overestimated substantially. Instead, we propose to count the number of actually accessed values. A single random read would increase the counter by one, whereas a full column scan would increment the counter by the number of values in the range (i. e., the block size). The access frequency of a value is then obtained by dividing the value range counter by the block size. The calculated access frequencies are nevertheless prone to skewed access patterns. More specifically, access frequencies of heavy hitters are underestimated, whereas frequencies of rarely accessed values (i. e., the long tail) are overestimated.

To improve precision in such situations, we propose to employ the space-saving algorithm and its stream-summary data structure [MAE05] in order to monitor the top- h most frequently accessed values of a value range. However, depending on h , not all values stored in the stream-summary are true heavy hitters. To identify actual heavy hitters from the values stored in the stream-summary, we additionally consider each values' value range counter. Since the stream-summary substantially overestimates access frequencies of rarely accessed values, we argue that the estimated frequency of a heavy hitter must not be significantly larger than its corresponding value range counter. Since the stream-summary also tends to overestimate heavy hitters, we tolerate a slightly larger estimated access frequency. Therefore, we introduce a tolerance parameter λ , such that the estimated access frequency of the stream-summary is only considered if its estimate is at most λ -times larger than its corresponding value range counter.

To calculate the access frequency of a value, we first check if the corresponding value range contains heavy hitters. If this is the case, we subtract their accumulated access count from the value range counter. The estimated access frequency of values from the long tail is given by the remaining block count divided by the number of values from the long tail in the value range. The estimated access frequency of heavy hitters is simply taken from the stream-summary.

Our approach can be tuned to fulfill the space requirement by configuring the block size and the number of heavy hitter candidates tracked by the stream-summary data structure. We show in Section 6 that our approach also achieves high precision while having a low runtime overhead. The presented data structure is summarized in the following:

Access Counter 4 (Table Partitioning Advisor)

Block Counting: For each attribute $A_{i,j} \in R_i$, we create counters $X_{i,j,0}^{val}, \dots, X_{i,j,b}^{val}, \dots, X_{i,j,\lfloor d_{i,j}/b_{i,j} \rfloor}^{val}$, where the block size $b_{i,j}$ is the number of values grouped into a block.

Stream-summary: For each attribute $A_{i,j} \in R_i$, we create a stream-summary data structure $SS_{i,j}^h$ such that $D(SS_{i,j}^h)$ is the domain of the monitored top- h most frequently accessed values. For a value $v_{i,j,k}$, the estimated access frequency is given by $SS_{i,j}^h(v_{i,j,k})$ if $v_{i,j,k} \in D(SS_{i,j}^h)$, otherwise 0.

Physical Accesses: We consider the physical data accesses during execution of workload W . For a sequential read on $A_{i,j}$, $X_{i,j,b}^{val}$ is incremented by the number of values that fall into

the given block and have at least one matching row. The values are also inserted into $SS_{i,j}^h$. For a random read $R_i[\text{rid}_i].A_{i,j} = v_{i,j,k}$, $\text{rid}_i \in [1, |R_i|]$, $X_{i,j,[k/b_{i,j}]}^{val}$ is incremented by 1, and the value is inserted into $SS_{i,j}^h$.

Access Frequency: The estimated access frequency $\hat{f}_{v_{i,j,k}}$ is calculated as follows:

$$\hat{f}_{v_{i,j,k}} = \begin{cases} SS_{i,j}^h(v_{i,j,k}) & \text{if } isHH(v_{i,j,k}) \\ \left\lceil \left(X_{i,j,[k/b_{i,j}]}^{val} - numHHAccesses \right) / (b_{i,j} - numHH) \right\rceil & \text{otherwise,} \end{cases}$$

$$\text{where } isHH(v_{i,j,k}) = \begin{cases} 1 & \text{if } v_{i,j,k} \in D(SS_{i,j}^h) \wedge SS_{i,j}^h(v_{i,j,k}) \leq \lambda \cdot X_{i,j,[k/b_{i,j}]}^{val} \\ 0 & \text{otherwise.} \end{cases}$$

$$numHH = \sum_{k'=\lfloor k/b_{i,j} \rfloor \cdot b_{i,j}}^{\lceil k/b_{i,j} \rceil \cdot b_{i,j}-1} isHH(v_{i,j,k'})$$

$$numHHAccesses = \sum_{k'=\lfloor k/b_{i,j} \rfloor \cdot b_{i,j}}^{\lceil k/b_{i,j} \rceil \cdot b_{i,j}-1} isHH(v_{i,j,k'}) \cdot SS_{i,j}^h(v_{i,j,k'}).$$

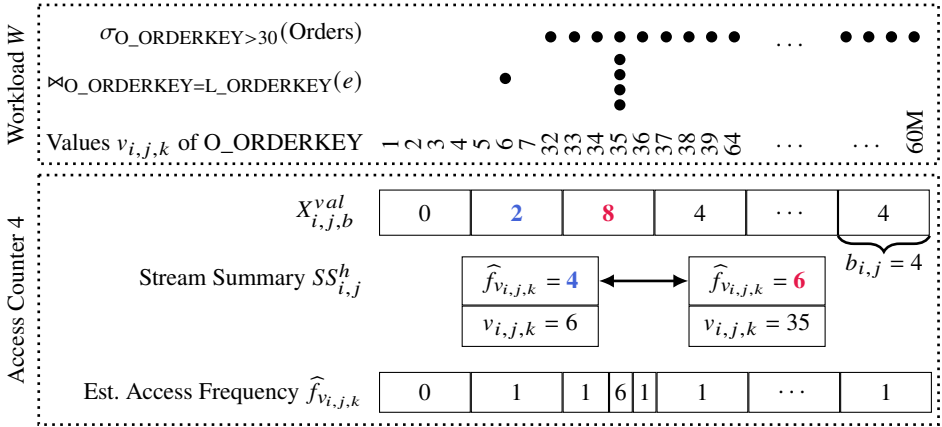


Fig. 7: Illustration of the data structure for collection *FStat F4* for a partitioning advisor.

Figure 7 shows for a selection and a join of attribute *O_ORDERKEY* how the access frequencies of values are estimated based on the block counter and the stream-summary. For example, the value 35 stored in the stream-summary is a heavy hitter as 6 is not larger than $\lambda \cdot X_{i,j,2}^{val}$ with $\lambda = 1.2$. Therefore, the counter $X_{i,j,2}^{val}$ is decremented by 6, which results in an estimated access frequency of 1 for the values from the long tail, i. e., 33, 34, and 36. In contrast, value 6 is not classified as a heavy hitter as the estimated access frequency 4 is more than λ -times larger than $X_{i,j,1}^{val}$ with $\lambda = 1.2$.

6 Experimental Evaluation

We evaluate the presented access counters with respect to their precision, space efficiency, and runtime overhead using real-world and synthetic benchmarks for an index advisor (Section 6.1), a data compression advisor (Section 6.2), a buffer pool size advisor (Section 6.3), and a table partitioning advisor (Section 6.4). We implemented our access counters prototypically in SAP HANA [MBL17]. First, we discuss the experimental setup.

Our test system is equipped with an Intel Xeon E7-8870 v4 CPU (4 sockets) and 1 TB DRAM. Secondary storage is provided by a RAID controller of 8 disks of type HGST HUC101812CSS204 HDD with 10k rpm and a SAS 12 Gbit/s interface.

The first workload is the synthetic TPC-H benchmark [TP18] with scale factor 10, consisting of 22 templated queries. To create a challenging environment for our access counters, we consider as second workload the JCC-H benchmark [BAK18] (scale factor 10), which extends TPC-H with data and query skew. For example, special shopping events such as Black Friday are reflected by corresponding spikes in `o_orderdate`. To cover the experiments in an acceptable time, we excluded queries Q9, Q16, Q20, and Q21 for JCC-H since parameter combinations led to query execution times larger than five minutes due to the data and query skew. Our third workload is the Join Order Benchmark (JOB) [Le15]. JOB consists of 33 different query templates (113 different queries in total) and uses real-world data from IMDb with data skew and correlations that aggravate estimation errors.

For the evaluation, we randomly generated for each benchmark a workload of 200 queries. The following table shows how often each templated query occurs in each workload:

Query ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
TPC-H	8	11	11	5	14	16	9	5	8	11	5	11	10	7	12	4	8	8	10	9	9	9											
JCC-H	11	15	7	14	19	9	9	12	–	7	14	10	12	8	9	–	9	14	10	–	–	11											
JOB	5	6	4	4	7	11	4	5	5	3	6	2	9	4	6	11	12	16	17	7	4	9	5	5	5	7	3	8	5	3	5	7	10

6.1 Use Case 1: Index Advisor

We start by evaluating Access Counter 1 for collecting workload execution statistics for an index advisor. Since we group actual output cardinalities into intervals $[b^r, b^{r+1})$ and count only the number of selections per interval, we calculate the precision of our approach by dividing the estimated output cardinality (i. e., $\sqrt{b^r \cdot b^{r+1}}$) by its actual output cardinality: $\varphi_{idx} = |\sigma_p(R_i)| / |\sigma_p(R_i)|$. In our experiments, we set the interval base parameter b to 2. Hence, the actual and recorded output cardinalities differ at most by a factor of $\sqrt{2}$.

Figure 8 shows for six attributes $\mathbb{A}_{i,s} \subseteq \mathcal{F}(p), \forall \sigma_p(R_i) \in T(q), \forall q \in W$ of each benchmark the precision φ_{idx} , i. e., the ratio of estimated and actual output cardinalities. Overestimation is shown on the top, underestimation at the bottom. Each boxplot shows the 0.00, 0.25, 0.5,

0.75, and 1.00 percentiles. We observe for all attributes and all benchmarks that φ_{idx} of all selections is at most $\sqrt{2}$ in accordance with our choice of b .

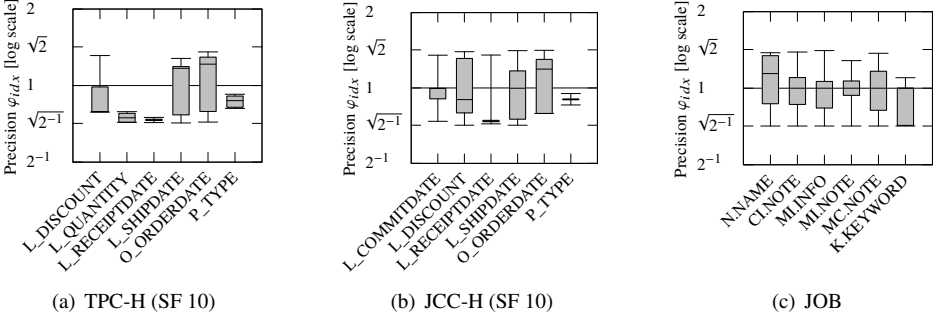


Fig. 8: Precision of our approach for collecting workload execution statistics for an index advisor.

Workload	Precise Counting			Our Approach		
	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB
Precision φ_{idx}	1.0	1.0	1.0	$\leq \sqrt{2}$	$\leq \sqrt{2}$	$\leq \sqrt{2}$
Memory Overhead	10.6%	10.6%	8.4%	$< 0.1\%$	$< 0.1\%$	$< 0.1\%$
Runtime Overhead	1.4%	1.5%	1.6%	1.7%	2.6%	3.1%

Tab. 4: Precision, space efficiency, and runtime overhead compared to precise counters.

Table 4 shows the results with respect to precision, space efficiency, and runtime overhead of precise counting (i. e., one counter per output cardinality) and our approach (i. e., lazy counters and interval counting). While precise counting achieves perfect precision, its memory overhead varies between 8.4% and 10.6% and is thus substantial. Our approach instead still attains reasonably accurate estimates, differing at most by a factor of $\sqrt{2}$. The memory overhead is also negligible due to lazy counting in combination with intervals. Both approaches yield a low runtime overhead since only the actual output cardinalities of selections are tracked. We conclude that our access counters are precise, compact, and fast.

6.2 Use Case 2: Data Compression Advisor

We now evaluate Access Counter 2 for collecting workload execution statistics for a data compression advisor. Table 5 shows the results with respect to precision, space efficiency, and runtime overhead. Our approach is 100% precise since, for each attribute, the exact number of rows accessed sequentially

Workload	TPC-H	JCC-H	JOB
Precision	100% precise		
Memory Overhead	$< 0.1\%$	$< 0.1\%$	$< 0.1\%$
Runtime Overhead	4.7%	8.3%	9.1%

Tab. 5: Precision, space efficiency, and runtime overhead for our access counters of a data compression advisor.

and randomly by the workload is counted. Maintaining just two 64-bit integer counters per attribute is also space-efficient. For example, for the Join Order Benchmark with 108 attributes in 21 relations, the total memory consumption is only 1.73 KB ($= 108 \cdot 16$ bytes). Compared to the data set size in SAP HANA (2.28 GB), this represents only 0.00008%. As the runtime overhead is also low (between 4.7% and 9.1%), we conclude that our access counters for a data compression advisor are precise, compact, and fast.

6.3 Use Case 3: Buffer Pool Size Advisor

In the third experiment, we evaluate Access Counter 3 for collecting workload execution statistics for a buffer pool size advisor. Table 6 shows the results with respect to the precision, space efficiency, and runtime overhead of naïve block-level counting (i. e., updating the frequencies of all touched pages) and our approach (i. e., updating only the frequencies of start and end pages). Both approaches are 100% precise since, for each memory page, all physical accesses are tracked. Compared to the data set size, the memory overhead is at most 0.2% compared to the tables data size, given the smallest page size of 4 KB in SAP HANA (64 bit/4 KB) [Sh19]. We use one signed 64-bit integer counter per page as counters may become negative. The runtime overhead of naïve block-level counting varies between 8.3% and 21.8%. Our approach results only in a runtime overhead between 5.2% and 13.5%, as updates to the counter are done in constant time for queries that span multiple pages. We conclude that our access counters are precise, compact, and fast.

Workload	Naïve Block-Level Counting			Our Approach		
	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB
Precision	100% precise			100% precise		
Memory Overhead	$\leq 0.2\%$	$\leq 0.2\%$	$\leq 0.2\%$	$\leq 0.2\%$	$\leq 0.2\%$	$\leq 0.2\%$
Runtime Overhead	8.3%	13.1%	21.8%	5.2%	9.2%	13.5%

Tab. 6: Precision, space efficiency, and runtime overhead compared to naïve block access counters.

6.4 Use Case 4: Table Partitioning Advisor

Finally, we evaluate Access Counter 4 for collecting workload execution statistics for a table partitioning advisor. To fulfill the space efficiency requirement, we limit the access counters' memory footprint to 1% compared to the column size (encoded column and dictionary). For example, for O_ORDERDATE (23 MB, 2406 distinct values), we create one counter per domain value, while for O_ORDERKEY (105MB, 15,000,000 distinct values), domain values are grouped into ranges of 115 values each. We also maintain a stream-summary for attributes with a block size larger than one to track the top-100 most frequently accessed values. Finally, we set $\lambda = 1.2$, i. e., a value is classified as heavy hitter if its access frequency estimated by the stream-summary is at most $1.2\times$ larger than its value range counter. We experimentally evaluated $\lambda = 1.2$ as a good choice. To calculate the

precision of a value $\varphi_{i,j,k}$, we divide the estimated access frequency by the actual access frequency, i. e., $\varphi_{i,j,k} = \hat{f}_{v_{i,j,k}} / f_{v_{i,j,k}}$.

In the JCC-H benchmark, 29 of 61 attributes yield a block size larger than one, i. e., cannot grant 100% precision within a memory budget of 1% of the column size. Figure 9 shows the precision $\varphi_{i,j,k}$ of three approaches and six representative attributes with a block size larger than one. Overestimation is shown on the top, underestimation at the bottom. The boxplot displays the 0.0001, 0.25, 0.5, 0.75, and 0.9999 percentiles. Outliers are highlighted as dots above or below the boxplot.

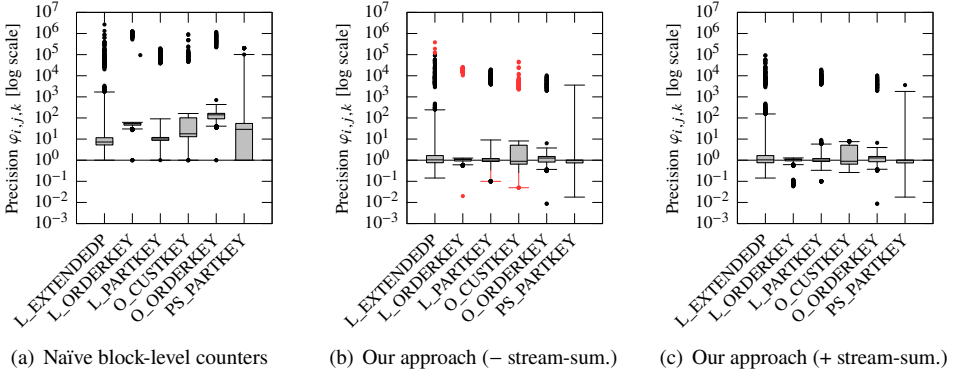


Fig. 9: Precision of our approach (with and without the stream-summary data structure) compared to naïve block-level access counters for a table partitioning advisor, executing the JCC-H benchmark.

Figure 9(a) shows the precision of naïve block-level counters, i. e., the block counter is incremented by one whenever a value or sub-range of the block is read. The results confirm the statement in Section 5.4 that access frequencies are overestimated substantially.

Figure 9(b) shows the precision of our approach that counts the number of actually accessed block values, while the access frequency of a value is obtained by dividing the total number of accessed values by the block size. We observe that our approach dramatically improves precision by several orders of magnitude, most of the estimates are within a bound of factor 2. However, for all six attributes, heavy hitters are underestimated, and rarely accessed values are overestimated (shown on the top and bottom of Figure 9(b)).

Figure 9(c) shows the precision obtained by adding a stream-summary to identify heavy hitters. To emphasize the difference with and without the stream-summary, we mark these values in Figure 9(b) in red, which are estimated correctly in Figure 9(c). For example, the heavy hitters of L_ORDERKEY (shown in red at the bottom in Figure 9(b)) are estimated precisely in Figure 9(c). Accordingly, rarely accessed values of the corresponding block are overestimated without the stream-summary (shown at the top of Figure 9(b)) but

estimated precisely with the stream-summary. We observe similar results for O_CUSTKEY, L_PARTKEY, and L_EXTENDEDPRICE.

We omit measurements of the precision for the TPC-H benchmark since the results are very similar compared to the JCC-H benchmark by ignoring the heavy hitters.

In the Join Order Benchmark, 47 of 108 attributes yield a block size larger than one. Figure 10 shows the precision $\varphi_{i,j,k}$ for six representative attributes. We again observe that naïve block-level counters overestimate access frequencies substantially (Figure 10(a)), while our approach improves the precision by 1-2 orders of magnitude (Figure 10(b)). However, we do not observe substantial improvement by adding a stream-summary like for the JCC-H benchmark (Figure 10(c)). The reason is that the JCC-H benchmark exhibits heavy hitters by design, while the Join Order Benchmark exposes only limited data and query skew.

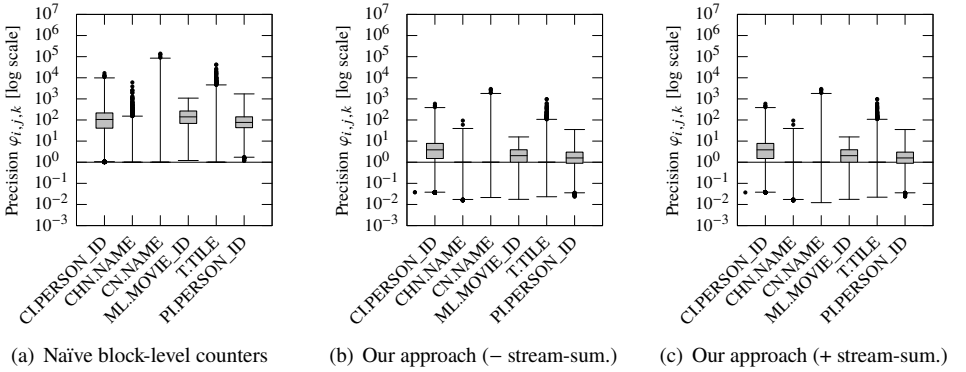


Fig. 10: Precision of our approach (with and without the stream-summary data structure) compared to naïve block-level access counters for a table partitioning advisor, executing the Join Order Benchmark.

Table 7 shows the space efficiency and runtime overhead of row-level access counters, naïve block-level access counters, and our approach, with and without the stream-summary data structure. While row-level data access counters are 100% precise, their memory overhead is high, and the runtime overhead is also notable. In contrast, naïve block-level access counters and our approach (without stream-summary) use a fixed memory budget of 1% and achieve

Workload	Row-Level Counters			Block-Level Counters & Our approach (– s.s.)			Our approach (+ stream summary)		
	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB	TPC-H	JCC-H	JOB
Memory Overhead	10.80%	10.82%	20.53%	$\leq 1\%$	$\leq 1\%$	$\leq 1\%$	$\leq 1\%$	$\leq 1\%$	$\leq 1\%$
Runtime Overhead	3.9%	14.7%	15.6%	2.1%	9.7%	9.6%	13.8%	22.7%	23.6%

Tab. 7: Memory and runtime overhead for our approach compared to row and block-level counters.

low runtime overhead. However, naïve block-level access counters are imprecise, while our approach achieves precise estimates (Figures 9 and 10). Adding the stream-summary data structure further improves the precision (Figure 9) at the cost of increasing the runtime overhead. Therefore, we argue that our approach (without the stream-summary) is preferred if the runtime overhead is critical. Otherwise, the stream-summary data structure may be added to improve the precision with low memory overhead.

7 Conclusion

We presented data structures that solve the problem of providing workload execution statistics with high precision, low memory consumption, and low runtime overhead to automated physical database design tools. Since no approach in the literature collects precise, compact, and fast workload execution statistics for an advisor tool, we presented how existing approaches can be combined and for which advisors new data structures have to be designed. Our evaluation showed that our data access counters outperform related work to provide precise, compact, and fast workload execution statistics for an index advisor, a data compression advisor, a buffer pool size advisor, and a table partitioning advisor using real-world and synthetic benchmarks.

References

- [ABI19] Athanassoulis, M.; Bøgh, K. S.; Idreos, S.: Optimal Column Layout for Hybrid Workloads. *Proc. VLDB Endow.* 12/13, pp. 2393–2407, Sept. 2019.
- [Ag04] Agrawal, S.; Chaudhuri, S.; Kollar, L.; Marathe, A.; Narasayya, V.; Syamala, M.: Database Tuning Advisor for Microsoft SQL Server 2005. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases. VLDB '04, VLDB Endow.*, pp. 1110–1121, 2004.
- [BAK18] Boncz, P.; Anatiotis, A.-C.; Kläbe, S.: JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In (Nambiar, R.; Poess, M., eds.): *Performance Evaluation and Benchmarking for the Analytics Era*. Springer International Publishing, Cham, Switzerland, pp. 103–119, 2018.
- [Cu10] Curino, C.; Jones, E.; Zhang, Y.; Madden, S.: Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3/1–2, pp. 48–57, Sept. 2010.
- [Da16] Das, S.; Li, F.; Narasayya, V. R.; König, A. C.: Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service. In: *Proceedings of the 2016 International Conference on Management of Data. SIGMOD '16*, Association for Computing Machinery, New York, NY, USA, pp. 1923–1934, 2016.

- [Da19] Damme, P.; Ungethüm, A.; Hildebrandt, J.; Habich, D.; Lehner, W.: From a Comprehensive Experimental Survey to a Cost-Based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.* 44/3, June 2019.
- [FKN12] Funke, F.; Kemper, A.; Neumann, T.: Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *Proc. VLDB Endow.* 5/11, pp. 1424–1435, July 2012.
- [Gu18] Gurajada, A.; Gala, D.; Zhou, F.; Pathak, A.; Ma, Z.-F.: BTrim: Hybrid in-Memory Database Architecture for Extreme Transaction Processing in VLDBs. *Proc. VLDB Endow.* 11/12, pp. 1889–1901, Aug. 2018.
- [Hu19] Huang, G.; Cheng, X.; Wang, J.; Wang, Y.; He, D.; Zhang, T.; Li, F.; Wang, S.; Cao, W.; Li, Q.: X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In: *Proceedings of the 2019 International Conference on Management of Data. SIGMOD '19*, Association for Computing Machinery, New York, NY, USA, pp. 651–665, 2019.
- [KAI17] Kester, M. S.; Athanassoulis, M.; Idreos, S.: Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe? In: *Proceedings of the 2017 ACM International Conference on Management of Data. SIGMOD '17*, Association for Computing Machinery, New York, NY, USA, pp. 715–730, 2017.
- [Ko20] Kossmann, J.; Halfpap, S.; Jankrift, M.; Schlosser, R.: Magic Mirror in My Hand, Which is the Best in the Land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13/12, pp. 2382–2395, July 2020.
- [Le10] Lemke, C.; Sattler, K.-U.; Faerber, F.; Zeier, A.: Speeding Up Queries in Column Stores. In (Bach Pedersen, T.; Mohania, M. K.; Tjoa, A. M., eds.): *Data Warehousing and Knowledge Discovery. Springer Berlin Heidelberg, Berlin, Heidelberg*, pp. 117–129, 2010.
- [Le15] Leis, V.; Gubichev, A.; Mirchev, A.; Boncz, P.; Kemper, A.; Neumann, T.: How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9/3, pp. 204–215, Nov. 2015.
- [LLS13] Levandoski, J. J.; Larson, P.-Å.; Stoica, R.: Identifying hot and cold data in main-memory databases. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE). ICDE '13*, IEEE, pp. 26–37, 2013.
- [Lu19] Lu, J.; Chen, Y.; Herodotou, H.; Babu, S.: Speedup Your Analytics: Automatic Parameter Tuning for Databases and Big Data Systems. *Proc. VLDB Endow.* 12/12, pp. 1970–1973, Aug. 2019.
- [MAE05] Metwally, A.; Agrawal, D.; El Abbadi, A.: Efficient Computation of Frequent and Top-k Elements in Data Streams. In (Eiter, T.; Libkin, L., eds.): *Database Theory - ICDT 2005. Springer Berlin Heidelberg, Berlin, Heidelberg*, pp. 398–412, 2005.

- [MBL17] May, N.; Böhm, A.; Lehner, W.: SAP HANA – The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In (Mitschang, B.; Nicklas, D.; Leymann, F.; Schöning, H.; Herschel, M.; Teubner, J.; Härder, T.; Kopp, O.; Wieland, M., eds.): *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*. Gesellschaft für Informatik, Bonn, pp. 545–546, 2017.
- [Na20] Nathan, V.; Ding, J.; Alizadeh, M.; Kraska, T.: Learning Multi-Dimensional Indexes. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. SIGMOD '20*, Association for Computing Machinery, New York, NY, USA, pp. 985–1000, 2020.
- [No20] Noll, S.; Teubner, J.; May, N.; Böhm, A.: Analyzing Memory Accesses with Modern Processors. In: *Proceedings of the 16th International Workshop on Data Management on New Hardware. DaMoN '20*, Association for Computing Machinery, New York, NY, USA, 2020.
- [Ra02] Rao, J.; Zhang, C.; Megiddo, N.; Lohman, G.: Automating Physical Database Design in a Parallel Database. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. SIGMOD '02*, Association for Computing Machinery, New York, NY, USA, pp. 558–569, 2002.
- [Se16] Serafini, M.; Taft, R.; Elmore, A. J.; Pavlo, A.; Aboulmaga, A.; Stonebraker, M.: Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* 10/4, pp. 445–456, Nov. 2016.
- [Sh19] Sherkat, R.; Florendo, C.; Andrei, M.; Blanco, R.; Dragusanu, A.; Pathak, A.; Khadilkar, P.; Kulkarni, N.; Lemke, C.; Seifert, S.; Iyer, S.; Gottapu, S.; Schulze, R.; Gottipati, C.; Basak, N.; Wang, Y.; Kandiyannallur, V.; Pendap, S.; Gala, D.; Almeida, R.; Ghosh, P.: Native Store Extension for SAP HANA. *Proc. VLDB Endow.* 12/12, pp. 2047–2058, Aug. 2019.
- [St06] Storm, A. J.; Garcia-Arellano, C.; Lightstone, S. S.; Diao, Y.; Surendra, M.: Adaptive Self-Tuning Memory in DB2. In: *Proceedings of the 32nd International Conference on Very Large Data Bases. VLDB '06*, VLDB Endowment, pp. 1081–1092, 2006.
- [TP18] TPC: TPC Benchmark H Standard Specification, http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf, 2018.

Exploring Memory Access Patterns for Graph Processing Accelerators

Jonas Dann¹, Daniel Ritter¹, Holger Fröning²



Abstract: Recent trends in business and technology (e. g., machine learning, social network analysis) benefit from storing and processing growing amounts of graph-structured data in databases and data science platforms. FPGAs as accelerators for graph processing with a customizable memory hierarchy promise solving performance problems caused by inherent irregular memory access patterns on traditional hardware (e. g., CPU). However, developing such hardware accelerators is yet time-consuming and difficult and benchmarking is non-standardized, hindering comprehension of the impact of memory access pattern changes and systematic engineering of graph processing accelerators.

In this work, we propose a simulation environment for the analysis of graph processing accelerators based on simulating their memory access patterns. Further, we evaluate our approach on two state-of-the-art FPGA graph processing accelerators and show *reproducibility*, *comparability*, as well as the shortened development process by an example. Not implementing the cycle-accurate internal data flow on accelerator hardware like FPGAs significantly reduces the implementation time, increases the benchmark parameter transparency, and allows comparison of graph processing approaches.

Keywords: DRAM; FPGA; Graph processing; Irregular memory access patterns; Simulation

1 Introduction

Recently, areas in computer science like machine learning, computational sciences, medical applications, and social network analysis drove a trend to represent, store, and process structured data as graphs [Be19, DRF20]. Consequently, graph processing gained relevance in the fields of non-relational databases and analytics platforms. As a possible solution to the performance problems on traditional hardware (e. g., CPUs) caused by irregular memory accesses and little computational intensity inherent to graph processing [Be19, DRF20, Lu07], FPGA accelerators emerged to enable unique memory access pattern and control flow optimizations [DRF20]. FPGAs, compared to CPUs or GPUs with their fixed memory hierarchy, have custom-usable on-chip memory and logic resources that are not constrained to a predefined architecture. Example 1 illustrates the effect of irregular memory accesses for breadth-first search (BFS) with an edge-centric approach. When not reading sequentially from DRAM, bandwidth degrades quickly [Dr07], due to significant latency introduced by DRAM row switching and partially discarded fetched cache lines.

¹ SAP SE, {firstname.lastname}@sap.com

² Heidelberg University, holger.froening@ziti.uni-heidelberg.de

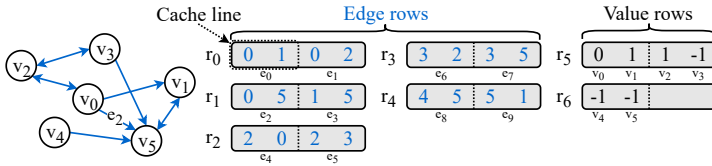


Fig. 1: Illustration of irregular memory accesses for breadth-first search

Example 1. Let each cache line consist of two values, the current BFS iteration be 1 with root v_0 , and e_2 be the current edge to be processed. Figure 1 shows an example graph with a simplified representation in DRAM memory. The graphs edge array is stored in rows r_0 – r_4 and the current value array is stored in r_5 and r_6 . We begin by reading edge e_2 which incurs activating r_1 in the memory and reading a full cache line. Then, we activate r_5 and read the first cache line containing v_0 and v_1 , but only use v_0 . Finally, we activate r_6 to read v_5 and write the new value 1 to the same location, while wasting bandwidth of one value on each request (i. e., reading and not writing the value of v_4 respectively).

While FPGA-based graph processing accelerators show good results for irregular memory access pattern acceleration (e. g., [Ya18, Zh19]), programming FPGAs is time-consuming and difficult compared to CPUs and GPUs where the software stack is much better developed [Ab19, BRS13]. Additionally, software developers currently lack the skill-set needed for high-performance FPGA programming, making development even more cumbersome. Aside from that, there are deficiencies in benchmarking of graph processing accelerators due to a large number of FPGAs on the market (almost every article uses a different FPGA), but also lack of accepted benchmark standards (cf. [DRF20]). This leads us to the two main challenges in the field: (1) time-consuming and difficult development of accelerators for irregular memory access patterns of graph processing, (2) differences in hardware platforms and benchmark setups hindering reproduction and comparison.

To solve challenges (1) and (2), we propose a simulation environment for graph processing accelerators (based on the idea in Fig. 2a) as a methodology and tool to quickly reproduce and compare different approaches in a synthetic, fixed environment. On a real FPGA, the on-chip logic implements data flow on on-chip (in block RAM (BRAM)) and off-chip state and graph data in the off-chip DRAM. Based on the observation that the access to DRAM is the dominating factor in graph processing, we however only implement an approximation of the off-chip memory access pattern in our environment working on the graph and state independently of the concrete (difficult to implement) data flow on the FPGA and feed that into a DRAM simulator. While the performance reported by such a simulation may not perfectly match real performance measurements, we see a high potential to better understand graph processing accelerators. This results in the following hypothesis:

Hypothesis. *Memory access patterns dominate the overall runtime of graph processing such that disregarding the internal data flow results in a reasonable error of a simulation.*

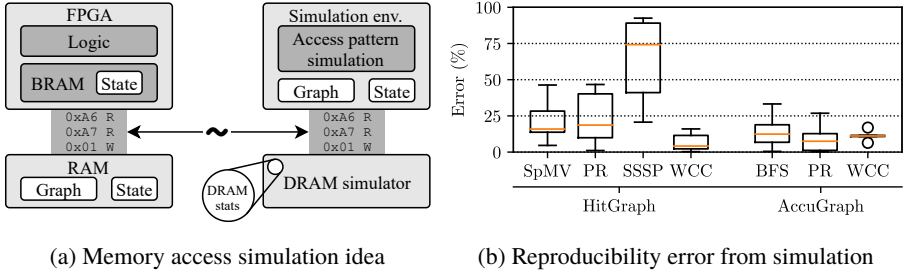


Fig. 2: Graph processing memory simulation idea and achieved reproducibility error

Our simulation approach significantly reduces the time to test new graph processing accelerator ideas and also enables design support and deeper inspection with DRAM statistics as well as easy parameter variation. In a recent survey [DRF20], we found multiple graph processing accelerator approaches (e. g., AccuGraph [Ya18], ForeGraph [Da17], HitGraph [Zh19], and Zhang et al. [ZL18] among others). Based on criteria like reported performance numbers on commodity hardware and sufficient conceptual details, we chose two state-of-the-art systems – namely HitGraph [Zh19] and AccuGraph [Ya18] – with orthogonal approaches representing the currently most relevant paradigms, edge- and vertex-centric graph processing, and evaluate our approach on their concepts. Figure 2b shows box plots of the percentage error $e = \frac{100 \times |s - t|}{t}$ we achieve for simulation performance s and ground truth performance t (taken from the respective article) grouped by accelerators and algorithms. Without single-source shortest-paths (SSSP) on HitGraph, we get a reasonable mean of 14.32%. We explain why this single algorithm performs so much worse and why there are outliers for AccuGraphs weakly-connected components (WCC) algorithm in Sect. 4.1. In this work, we make the following contributions:

1. We propose a *simulation environment* for graph processing accelerator engineering and *memory access abstractions* based on our hypothesis.
2. We conduct a *comprehensive reproducibility study* for the two representative graph processing accelerators HitGraph and AccuGraph and uncover deficiencies in performance measurement practices.
3. We show the reduced effort of engineering new ideas with our simulation environment by example of *two novel optimizations to AccuGraph*.

This article is structured as follows. In Sect. 2 we introduce basic concepts of graph processing, FPGA-addressable DRAM, and DRAM simulation. In Sect. 3 we conceptually specify the simulation environment, request flow abstractions, and their application to HitGraph and AccuGraph. In Sect. 4 we reproduce and compare the performance measurements of HitGraph and AccuGraph. We show the engineering benefits of our approach in Sect. 5, before discussing related work in Sect. 6 and concluding in Sect. 7.

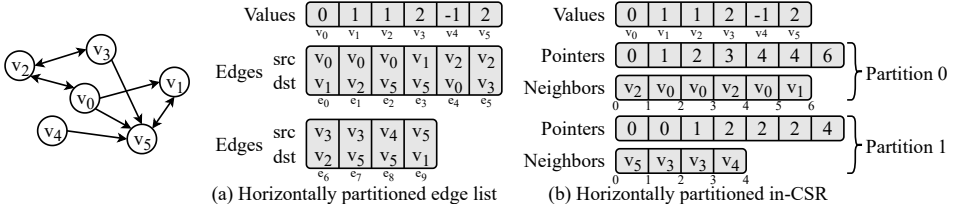


Fig. 3: Graph partitioning and data structures

2 Background

We start this section by briefly specifying graphs, how graph processing can be implemented, and what problems can be solved on graphs. Thereafter, we shortly introduce the memory hierarchy of FPGAs and more specifically how DRAM works internally. Lastly, we motivate the selection of Ramulator [KYM16] as our DRAM simulator and briefly explain how Ramulator models memory and is configured for our purpose.

2.1 Graph Processing

A graph $G = (V, E)$ is an abstract data structure consisting of a vertex set V and an edge set $E \subseteq V \times V$. Intuitively, they are used to describe a set of entities (vertices) and their relations (edges). Figure 3 shows two possible data structure representations (both with two partitions) of the example graph. Horizontally partitioned means dividing up the vertex set of the graph into intervals and assigning edges to the partition which interval contains their source vertex. Figure 3a shows the example graph as a horizontally partitioned edge list (used by HitGraph [Zh19]), which stores the graph as arrays of edges with a source and a destination vertex. For example, edge e_0 connects source v_0 to destination vertex v_1 . Figure 3b shows the same graph as a horizontally partitioned compressed sparse row (CSR) format of the inverted edges (used by AccuGraph [Ya18]), meaning all source and destination vertices of the edges in E are swapped before building a CSR data structure on them. CSR is a data structure for compressing sparse matrices (in this case the adjacency matrix of the graph) with two arrays. The values of the pointers array at position i and $i + 1$ delimit the neighbors of v_i stored in the neighbors array. For example, for v_5 in partition 1 the neighbors are the values of the neighbors array between 2 and 4, i. e., v_3 and v_4 .

Depending on the underlying graph data structure, graphs are processed based on two fundamentally different paradigms: edge- and vertex-centric graph processing. Edge-centric systems (e. g., HitGraph) iterate over the edges as primitives of the graph on an underlying edge list. Vertex-centric systems iterate over the vertices and their neighbors as primitives of the graph on an underlying adjacency list (e. g., CSR). Further, for the vertex-centric paradigm, there is a distinction into push- and pull-based data flow. A push-based data flow denotes that values are pushed along the forward direction of edges to update neighboring vertices. A pull-based data flow (e. g., applied by AccuGraph) denotes that values are pulled along the inverse direction of edges from neighboring vertices to update the current vertex.

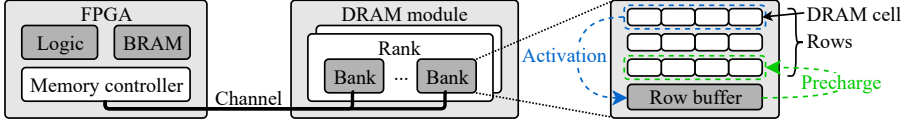


Fig. 4: DRAM (adapted from [KYM16])

In the context of this article, we consider the five graph problems implemented by HitGraph and AccuGraph: BFS, SSSP, WCC, sparse matrix-vector multiplication (SpMV), and PageRank (PR). The problems specify their implementations to varying degrees. For example, BFS denotes a sequence of visiting the vertices of a graph. Starting with a root vertex as the frontier, in each iteration, every unvisited neighbor of the current frontier vertices is marked as visited, assigned the current iteration as its value, and added to the frontier of the next iteration.

In contrast, SSSP only specifies the desired output, i. e., for each vertex $v \in V$ the shortest distance to the root vertex. The shortest distance equals the smallest sum of edge weights of any path from the root to v . If every edge is assumed to have weight 1, the result is equal to BFS. Similarly, WCC specifies as output for each vertex its affiliation to a weakly-connected component. Two vertices are in the same weakly-connected component if there is an undirected path between them. There is no requirement on how these outputs are generated.

Finally, SpMV and PR specify the execution directive. SpMV multiplies a vector (equal to V) with a matrix (equal to E) in iterations. PR is a measure to describe the importance of vertices in a graph. It is calculated by recursively applying $p(i) = \frac{1-d}{|V|} + \sum_{j \in N_G(i)} \frac{p(j)}{d_G(j)}$ for each $i \in V$ with damping factor d , neighbors N_G and degree d_G .

2.2 Memory Hierarchies of Field Programmable Gate Arrays

As a processor architecture platform, FPGA chips map custom architecture designs (i. e., a set of logic gates and their connection) to a grid of resources (e. g., look-up tables, flip-flops, and block RAM (BRAM)) connected with a programmable interconnection network. The memory hierarchy of FPGAs is split up into on-chip and off-chip memory. On-chip, FPGAs contain BRAM in the form of SRAM memory components. On modern FPGAs, there is about as much BRAM as there is cache on modern CPUs (all cache levels combined), but contrary to the fixed cache hierarchies of CPUs, BRAM is memory finely configurable to the application. For storage of larger data structures, DRAM (e. g., DDR3 or DDR4) is attached as off-chip memory. Subsequently, we briefly introduce the internal structure of DDR3 and DDR4 to understand its implications on graph processing.

The internal organization of DDR3 memory is shown in Fig. 4, which at the lowest level contains DRAM cells each representing one bit. The smallest number of DRAM cells

JESD79-3 DDR3 SDRAM Standard

JESD79-4 DDR4 SDRAM Standard

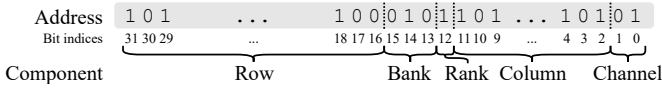


Fig. 5: DRAM addressing

(e. g., 16) that is addressable is called a column. Several thousand (e. g., 1, 024) columns are grouped together into rows. Further, independently operating banks combine several thousand (e. g., 65, 536) rows with a row buffer each.

Requests to data in a bank are served by the row buffer based on three scenarios: (1) When the addressed row is already buffered, the request is served with low latency (e. g., t_{CL} : 11ns). (2) If the row buffer is empty, the addressed row is first activated (e. g., t_{RCD} : 11ns), which loads it into the row buffer, and then the request is served. (3) However, if the row buffer currently contains a different row from a previous request, the current row has to be first pre-charged (e. g., t_{RP} : 11ns) and only then the addressed row can be activated and the request served. Additionally, there is a minimum latency between switching rows (e. g., t_{RAS} : 28ns). Thus, for high performance, row switching should be minimized.

Since one bank does not provide sufficient bandwidth, 8 parallel banks further form a rank. Multiple ranks operate in parallel but on the same I/O pins, thus increasing capacity of the memory, but not bandwidth. Finally, the ranks of the memory are grouped into channels. Each channel has its own I/O pins to the FPGA such that the bandwidth linearly increases with the number of channels. DDR4 contains another hierarchy level called bank groups, which group two to four banks to allow for more rapid processing of commands.

Data in DRAM is accessed by giving the memory a physical memory address that is split up into multiple parts internally representing addresses for each component in the DRAM hierarchy (cf. Fig. 5). Based on this, different addressing schemes are possible. An example addressing scheme that aids distribution of requests over channels might first address the channels, meaning subsequent addresses go to different channels, then address columns, ranks, banks, and rows. To further improve memory bandwidth, modern DRAM returns multiple bursts of data for each request (also called prefetching). For DDR3 and DDR4, each request returns a total of 64 Bytes over 8 cycles which we call a cache line in the following.

2.3 DRAM Simulators – Ramulator

To speed up the engineering of graph processing on FPGA accelerators, a DRAM Simulator is an integral part of our simulation environment (cf. Fig. 2a). For our purposes we need a DRAM simulator that supports DDR3 (for HitGraph [Zh19]) and DDR4 (for AccuGraph [Ya18]). We chose Ramulator [KYM16] for this work over other alternatives like DRAMSim2 [RCJ11] and USIMM [Ch12] because – to the best of our knowledge – it is the only DRAM simulator which supports (among many others like LPDDR3/4 and HBM) both of those DRAM standards (DDR3/4).

Ramulator models DRAM as a tree of state machines (e. g., channel, rank, bank in DDR3) where transitions are triggered by user or internal commands. However, Ramulator does not make any assumptions about data in memory. Purely the request and response flow is modelled with requests flowing into Ramulator and responses being called back. The Ramulator configuration parameters that are relevant to our work are: (1) DRAM standard, (2) channel count, (3) rank count, (4) DRAM speed specification, (5) DRAM organization.

3 Memory Access Simulation Environment

In this section, we first introduce the simulation environment based on the implications of our hypothesis and show the abstractions we developed to implement memory access patterns. Thereafter, we show how this can be applied to real graph processing accelerators. As motivated in Sect. 1, we chose HitGraph [Zh19] and AccuGraph [Ya18] for that.

3.1 Simulation Environment

As we established in Sect. 1, one of the main challenges with evaluating new graph processing ideas on FPGAs is time-consuming and difficult development of the accelerator. Thus, the goal of our simulation environment is reducing development time and complexity within reasonable error when compared to performance measurements on hardware. To achieve this goal we relax the necessity of cycle accurate simulation of on-chip data flow due to our hypothesis: *Memory access patterns dominate the overall runtime of graph processing such that disregarding the internal data flow results in a reasonable error of a simulation.* Modelling the off-chip memory access pattern means modelling request types, request addressing, request amount, and request ordering. Request type modelling is trivial since it is clear when requests read and write data. For request addressing, we assume that the different data structures (e. g., edge list and vertex values) lie adjacent in memory as plain arrays. We generate memory addresses according to this memory layout and the width of the arrays types in Bytes. Request amount modelling is mostly based on the size n of the vertex set, the size m of the edge set, average degree deg , and partition number p . We only simulate request ordering through mandatory control flow caused by data dependencies of requests. We assume that computations and on-chip memory accesses are instantaneous by default. In the following we introduce memory abstractions we developed for modelling request and control flow.

Figure 6 shows an overview of the memory access abstractions and their icons grouped by their role during memory access as *producer*, *merger*, and *mapper*.

Producer At the start of each request stream, a *producer* (Fig. 6a) is used to turn control flow (dashed arrow) triggers into a request stream (solid arrow). The producer might be rate limited, but if only a single producer is working at a time or requests are load balanced down-stream, the requests are just created in bulk.

Mergers Multiple request streams might then be merged with *mergers*, since Ramulator only has one endpoint. We have deduced abstractions to merge requests in a *direct* (Fig. 6b),

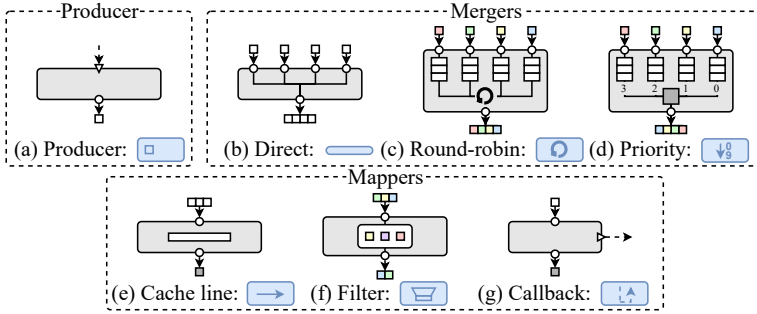


Fig. 6: Memory access abstractions

round-robin (Fig. 6c), and *priority*-based (Fig. 6d) fashion. If there are multiple request streams that do not operate in parallel, direct merging is applied. If request streams should be equally load-balanced, round-robin merging is applied. If request streams should take precedence over each other, priority merging is applied. For this, a priority is assigned to each request stream and requests are merged based on that.

Mappers Additionally to request creation with producers and ordering with mergers, we also found abstractions for request *mappers*. Thus, we introduce *cache line* buffers (Fig. 6e) for sequential or semi-sequential accesses that merge subsequent requests to the same cache line into one request. We do buffering such that multiple concurrent streams of requests benefit from it independently by placing it as far from the memory as necessary to merge the most requests. For data structures that are placed partially in on-chip memory (e. g., prefetch buffers and caches), and thus partially not require off-chip memory requests, we introduce request *filters* (Fig. 6f) that discard filtered requests. For control flow, we use a *callback* (Fig. 6g) abstraction. We disregard any delays in control flow propagation and just directly let the memory call back into the simulation. If requests are served from a cache line or filter abstraction, the callback is executed, if it is present.

In our simulation environment we instantiate a graph processing simulation and a Ramulator instance, and tick them according to their respective clock frequency. For graph processing simulation we focus on configurability of all aspects of the simulation such that we can quickly run differently parameterized performance measurements. Our simulation works on multiple request streams that are merged into one and fed into Ramulator. This leads us to a immensely reduced implementation time and complexity, gives us more insight into the systems, and provides portability of ideas developed in the simulation environment.

3.2 HitGraph

HitGraph [Zh19] is an edge-centric graph processing accelerator that claims to be among the best performing systems. The basic idea is to partition the graph horizontally into p partitions, stored as edge lists (cf. Sect. 2.1), and process the partitions in two phases in each iteration. First, updates are produced for each edge in each partition in the scatter phase.

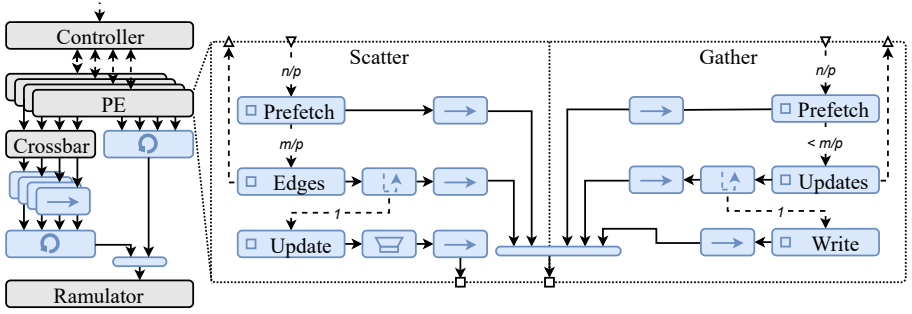


Fig. 7: HitGraph request and control flow

Second, all updates are applied to their respective vertex for each partition in the gather phase. The main goal behind this approach is to completely eliminate random reads to data and largely reduce the amount of random writes such that only semi-random writes remain. All reads to values of vertices are served from the prefetched partition in BRAM and all reads to either edges or updates are sequential. Writing updates is sequential, while writing values is the only semi-random memory access. Figure 7 shows the request and control flow modelling with our simulation environment. Execution starts with triggering a controller that itself triggers iterations of edge-centric processing until there were no changes to vertex values in the previous iteration. In each iteration, the controller first schedules all partitions for the scatter phase, before scheduling all partitions to the gather phase. Partitions are assigned beforehand to channels of the memory (four channels in [Zh19]) and there is a processing element (PE) for each channel. After all partitions are finished in the gather phase, the next iteration is started or the accelerator terminates.

Scatter The scatter phase starts by prefetching the $\frac{n}{p}$ values of the current partition into BRAM. Those requests go to a cache line abstraction, such that requests to the same cache line do not result in multiple requests to Ramulator. After all requests are produced, the prefetch step triggers the edge reading step that reads all $\frac{m}{p}$ edges of the partition. This is only an average value since the exact number of edges in a partition might vary because of skewed vertex degrees. For each edge request, we attach a callback that triggers producing an update request and merge them with a cache line abstraction. The update requests might be filtered by an optimization resulting in less than one update per edge. The target address depends on its destination vertex that can be part of any of the p partitions. Thus, there is a crossbar that routes each update request to a cache line abstraction for each partition, which sequentially writes it into a partition-specific update queue. After all edges have been read, the edge reader triggers the controller, which either triggers the next partition or waits on all memory requests to finish before switching phases.

Gather Similar to scatter, the gather phase starts with prefetching the $\frac{n}{p}$ vertex values sequentially. After value requests have been produced, the prefetcher triggers the update reader, which sequentially reads the update queue written by the scatter phase before. For

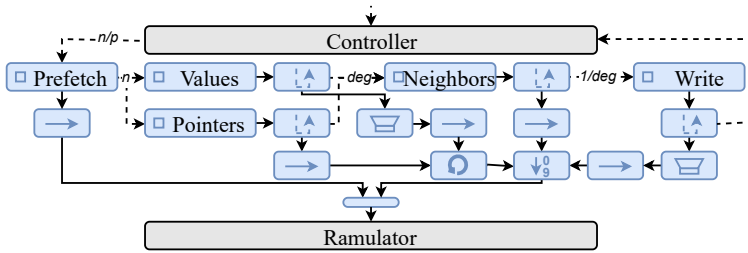


Fig. 8: AccuGraph request and control flow

each update we register a callback that triggers the value write. The value writes are not necessarily sequential, but especially for iterations where a lot of values are written, there might be a lot of locality. Thus, new values are passed through a cache line abstraction.

Parallelization All request streams in each PE are just merged directly into one stream without any specific merging logic, since mostly only one producer is producing requests at a time. However, edge and update reading is rate limited to the number of pipelines in each PE (which is set to 8 in the original article). Since all PEs are working on independent channels and Ramulator only offers one endpoint for all channels combined, we employ a round robin merge of the PE requests in order not to starve any channel. In addition, HitGraph applies optimizations to update generation. As a first step, the edges are sorted by destination vertex in each partition. This enables merging updates to the same destination vertex before writing them to memory, reducing the amount of updates u from $u = m$ to $u < n \times p$, and providing locality to the gather phases value writing. As a second optimization, an active bitmap with cardinality n is kept in BRAM that saves for each vertex if its value was changed in the last iteration. This enables update filtering, by filtering out updates from inactive vertices which saves a significant amount of update writes for most algorithm and data set combinations. As a final optimization, partitions with unchanged values or no updates are skipped, which saves time spent for prefetching of values and edge/update reading for some algorithms.

Configuration HitGraph is parameterized with the number of PEs p , pipelines q , and the partition size k . The number of PEs p is fixed to the number of memory channels because each PE works on exactly one memory channel. The pipeline count q is limited by the bandwidth available per channel given as the cache line size divided by the edge size. Lastly, the partition size is chosen such that $p \times m$ vertices fit into BRAM. HitGraph is able to use all available bandwidth due to fitting p and q to use all memory channels and whole cache lines of each channel per cycle. Hence, adding more compute (i. e., PEs or pipelines) would not help to solve the problem more efficiently which is in line with our hypothesis, i. e., memory access dominates the performance.

3.3 AccuGraph

AccuGraph [Ya18] is a vertex-centric graph processing accelerator with pull data flow. The basic idea is to partition the graph horizontally, store it as in-CSR data format and pull












updates from destination vertices (cf. Sect. 2.1). The original article proposes a flexible accumulator able to merge many updates to vertex values per cycle. Figure 8 shows the request and control flow modelling of AccuGraph. The controller is triggered to start the execution. It iterates over the graph until there are no more changes in the previous iteration. Each iteration triggers processing of all partitions. Partition processing starts with prefetching the $\frac{n}{p}$ source vertex values sequentially. Thereafter, values and pointers of all destination vertices are fetched. The value requests are filtered by the values that are already present in BRAM from the partition prefetching. Pointers are fetched purely sequentially. Those two request streams are merged round robin, because a value is only useful with the associated pointers. For every value fetched in this way, neighbors are read from memory sequentially. Since the neighbors of subsequent vertices are in sequence in CSR, this is fully sequential. An internal accumulator collects the changes caused through the neighbors and writes them back to memory, when all neighbors were read. The value changes are also directly applied to the values currently present in BRAM for a coherent view of vertex values. This is filtered such that only values that changed are written. All of these request streams are merged by priority, with write request taking the highest priority and neighbors the second highest because otherwise the computation pipelines would be starved. Additionally, we rate-limit neighbors loading to the number of edge pipelines present in the accelerator.

Configuration AccuGraph is parameterized by the number of vertex and edge pipelines (8 and 16 in the original article) and the partition size. Similar to HitGraph’s PE and pipeline fitting, the number of edge pipelines is specifically chosen to allow processing one cache line of edges per clock cycle and thus use the entire bandwidth of the memory, again in line with our hypothesis. The original article also describes an FPGA-internal data flow optimization which allows to approximate pipeline stalls, improving simulation accuracy significantly. The vertex cache used for the prefetched values is partitioned into 16 BRAM banks on the FPGA which can each serve one vertex value request per clock cycle. Since there are 16 edge pipelines in a standard deployment of AccuGraph, performance deteriorates quickly, when there are stalls. Thus, we implement stalls of this vertex cache in the control flow between the neighbors and write producers. A neighbors request callback is delayed until the BRAM bank can serve the value request.

4 Evaluation

In this section, we validate our simulation approach by reproducing the results reported for HitGraph [Zh19] and AccuGraph [Ya18], by indeed showing a reasonable simulation error compared to the measurements on real FPGA hardware. In addition, we illustrate for the first time, how these completely different graph processing approaches can be compared.

We take the same data sets (Table 1) and graph problems reported in the original articles to replicate their experiments. Only the two data sets live-journal and wiki-talk are used in both articles. HitGraph also measured performance on high diameter, constant degree graphs (i. e., roadnet-ca and berk-stan) and two instances of rmat synthetic graphs. AccuGraph measured performance on additional social graphs. Both selections of data sets contain

Name	Abbr.	Vertices	Edges	Dir.	Degs.	Avg.	ø	SCC
live-journal	lj	4, 847, 571	68, 993, 773	👍		14.23	16	0.790
wiki-talk	wt	2, 394, 385	5, 021, 410	👍		2.10	11	0.047
twitter	tw	41, 652, 230	1, 468, 364, 884	👍		35.25	75	0.804
rmat-24-16	r24	16, 777, 216	268, 435, 456	👍		16.00	19	0.023
rmat-21-86	r21	2, 097, 152	180, 355, 072	👍		86.00	14	0.103
roadnet-ca	rd	1, 971, 281	2, 766, 607	👎		2.81	849	0.993
berk-stan	bk	685, 231	7, 600, 595	👍		11.09	514	0.489
orkut	or	3, 072, 627	117, 185, 083	👎		76.28	9	1.000
youtube	yt	1, 157, 828	2, 987, 624	👎		5.16	20	0.980
dblp	db	425, 957	1, 049, 866	👎		4.93	21	0.744
slashdot	sd	82, 168	948, 464	👍		11.54	13	0.868

Abbr.: Abbreviation; Dir.: Directed; Degs.: Degree distribution on log. scale; Avg.: Average degree; ø: Diameter; SCC: Ratio of vertices in the largest strongly-connected component to n ; 👍: yes, 👎: no

Tab. 1: Graph data sets used by HitGraph and AccuGraph (all graphs from SNAP [LK14])

Approach	Type	Channels	Ranks	Speed	Organization
HitGraph	DDR3	4	2	1600K	8Gb_x16
AccuGraph	DDR4	1	1	2400R	4Gb_x16
Comparability	DDR4	1	1	2400R	8Gb_x16

Tab. 2: DRAM configurations

Approach	Weighted	SpMV	SSSP	PR	WCC	BFS	Vertex	Pointer
HitGraph	👍	32	32	32	32	-	32	-
AccuGraph	👎	-	-	32	32	8	32	32
Comparability	👎	-	32	32	32	32	32	32

Tab. 3: Data structure configurations (type width in bits)

directed graphs, while WCC only yields correct results for undirected graphs. This does not concern our reproducibility measurements but needs to be considered in the future.

4.1 Reproducibility

We measure the quality of the simulation as the percentage error $e = \frac{100 \times |s - t|}{t}$ of the simulation performance measurement s compared against the ground truth t reported by the respective article. The HitGraph numbers are extracted from a table and the AccuGraph numbers are taken from a chart. To reproduce the experiments as closely as possible, we parameterized the simulation environment according to configurations from the original articles. Table 2 shows the memory configurations of the reproducibility studies and the comparability study. Table 3 shows the data structure configurations. HitGraph uses weighted graphs and uniformly wide value types for all problems. AccuGraph uses unweighted graphs

Not officially listed on the SNAP [LK14] website anymore

Approach	PEs	Pipelines	Elements	Vertex pipelines	Edge pipelines	VS	ES
HitGraph	4	8	256, 000	-	-	-	-
AccuGraph	-	-	∞	8	16	8	8
Comparability	1	16	1, 024, 000	8	16	8	8

PEs: Processing elements; VS: Vertex pipeline size; ES: Edge pipeline size

Tab. 4: Parameter configurations

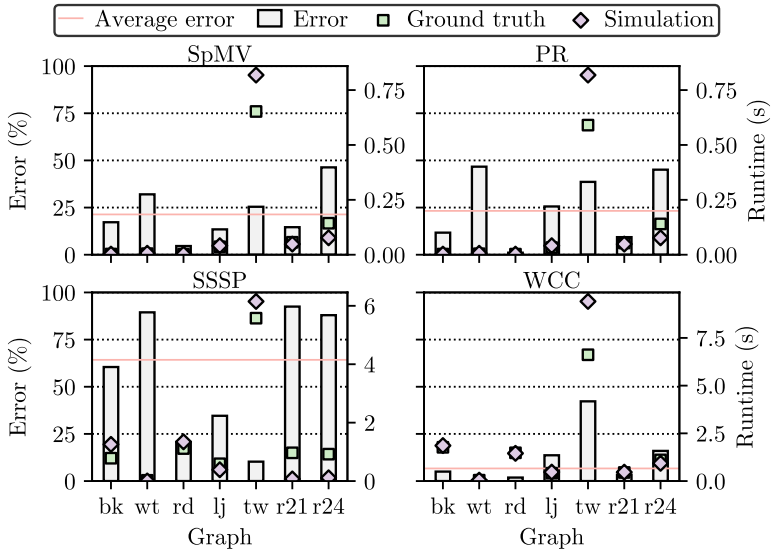


Fig. 9: HitGraph measurements

and an optimized 8 bits wide unsigned integer for BFS (problematic for constant degree graphs). Table 4 shows the respective graph processing accelerator parameters described in Sect. 3.1 and how they are configured. Both approaches share the partition size as a parameter. For the reproducibility study, AccuGraph is assumed to fit all vertices in BRAM for BFS and only for PR and WCC measurements on live-journal and orkut, the partition size is set to 1, 700, 000 vertices.

Figure 9 shows the HitGraph performance measurements for SpMV, PR, SSSP, and WCC as runtime in seconds (raw numbers can be found in Tab. 5). Overall, we observe a consistent outlier in the twitter graph. However, we notice that the HitGraph article reports the diameter of the twitter graph as being 15, while we report it as being 75 (cf. Tab. 1). Thus, we assume that our version of the graph is different and exclude it from all error averages in this article while still showing it in the plots for completeness (error source ①). SpMV and PR result in the same simulation performance, but since ground truth values are slightly different, we get a different error. In the original article, the authors measure only a single iteration of

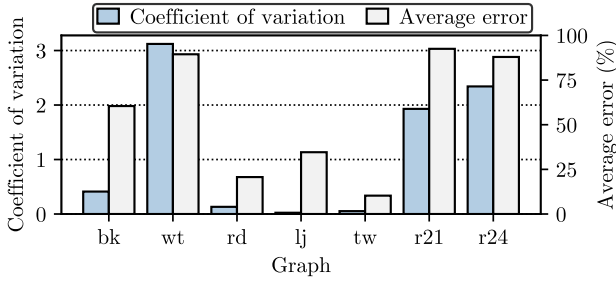


Fig. 10: HitGraph SSSP runtime variation study

SpMV and PR. However, we found that for very short runtimes of single iteration executions differences of a few cycles can already cause large deviations leading to the errors we observe for SpMV and PR (error source ②). We advise using multiple iterations of such algorithms in benchmarks in the future.

SSSP shows by far the worst error, with some simulations running much shorter in simulation than in the ground truth measurements. This can be explained by the problem’s dependence on the input root vertex (error source ③). The HitGraph authors randomly choose 20 root vertices and report the average runtime. However, wiki-talk and the rmat graphs have many strongly-connected components (SSCs) with just one or a few vertices (cf. Tab. 1). This causes algorithms like SSSP or BFS to immediately terminate after one iteration over the graph with very little runtime which results in large variation in performance measurements for root vertices from many small and few big SSCs shown in Fig. 10. The error is strongly correlated to the coefficient of variation in the runtimes (given by $\frac{\sigma}{\mu}$ with the standard deviation σ and the mean μ). This leads us to the conclusion that 20 random root vertices are not enough to stabilize the runtime measurements for graphs with such structure. We advocate for sharing how roots are picked in the future. Moreover, the HitGraph article does not specify how edge weights are set in the graph, which can also influence runtimes of SSSP (error source ④). We initialized all weights to 1. We regard WCC as the most reliable indicator for simulation quality because it does not depend on input variables and runs long enough so fixed overheads are irrelevant. We observe a low simulation error for WCC, which reassures us that the off-chip memory access modelling works well for HitGraph. Besides the twitter graph (which we explicitly excluded), the simulation almost perfectly matches the ground truth.

Figure 11 shows performance measurements for AccuGraph for BFS, PR, and WCC as billions of read edges per second (GREPS) (raw numbers can be found in Tab. 6). We calculate REPS as the number of actually read edges $m \times i$ (where i is the number of iterations) divided by the runtime r , which the original article calls traversed edges per second (TEPS). However, this is misleading since the well-known Graph500 benchmark

We generated the 20 random root vertices with the mt19937 generator in C++ with seed 3483584297.

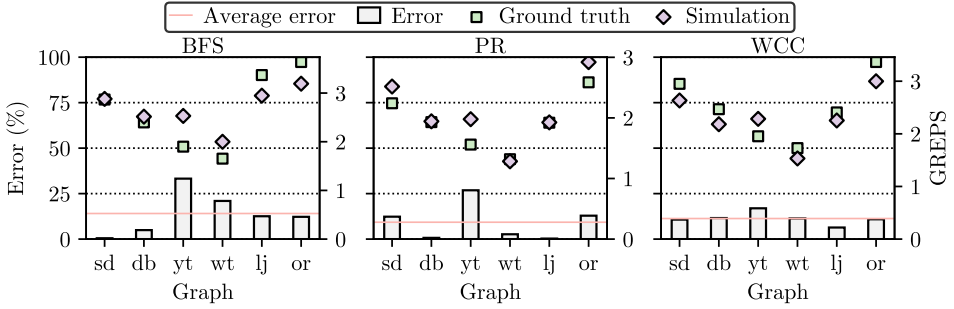


Fig. 11: AccuGraph measurements

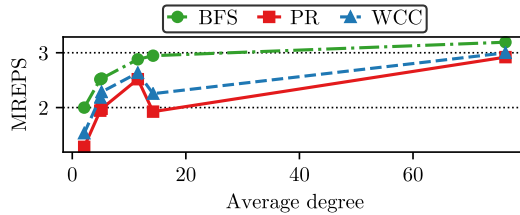


Fig. 12: AccuGraph performance by average degree

defines TEPS as the number of edges in a graph m divided by the runtime r . Thus, we rename the performance measure to REPS. As we already saw in Fig. 2b, the average error is very similar for all problems and fits the relative performance of the graph data sets well.

The only consistent outlier is the youtube graph which relatively performs better in all simulation measurements than is suggested by the ground truth measurements (error source ⑤). The original article notes that the performance of AccuGraph logarithmically depends on the average degree of vertices which we also reproduced (cf. Fig. 12). Thus, youtube should perform the way our measurements suggest, because it has a slightly higher average degree than the dblp graph. This may be an anomaly in the measurements performed by the AccuGraph authors. WCC is slightly slower in our simulations than they are on the accelerator and PR is slightly faster. There may be a fixed overhead that we are measuring in our experiments and is not measured in theirs. The better performance of PR, however, is expected, since we do not take the longer latencies and incurred pipeline stalls of floating point arithmetics into account (error source ⑥).

4.2 Comparability

With these encouraging reproducibility errors and the deeper insight in the approaches configurations, Fig. 13 shows a comparison of HitGraph and AccuGraph on an equal configuration (cf. Comparability in Tab. 2 – Tab. 4). It is not easily possible to use AccuGraph with weighted edges, such that we chose unweighted edges for these measurements. Also it

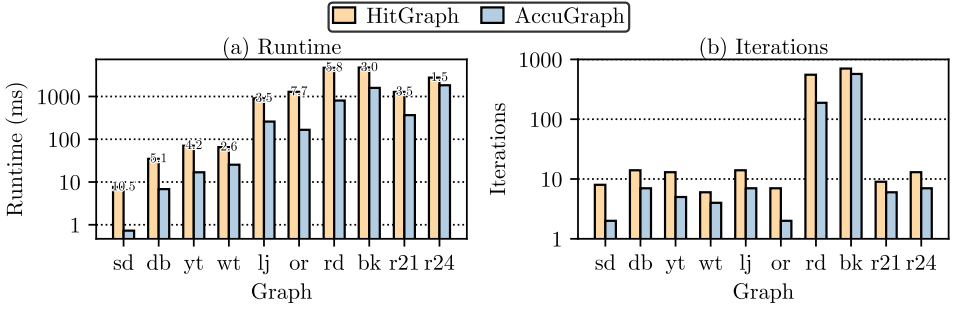


Fig. 13: HitGraph vs. AccuGraph on comparable configurations (with improvement of AccuGraph over HitGraph for runtime)

was not possible to expand AccuGraph to use four memory channels, such that we took the AccuGraph DRAM configuration, but increased the memory size to 8GB to be able to accommodate the rmat graphs. However, even this DRAM configuration does not fit the twitter graph which we thus excluded. Moreover, we configured HitGraph to process up to 16 edges each cycle just like AccuGraph and set the partition size to a reasonable 1,024,000 vertices. We show performance numbers of WCC on all graphs used in either of the original articles, since we got the lowest error for WCC. This includes high diameter graphs (e. g., roadnet and berkley-stanford) that AccuGraph has not been tested on yet.

For the two graphs that both systems were originally tested on, AccuGraph (~ 1728 MREPS) reported slightly higher numbers than HitGraph (1665 MREPS) on wiki-talk and HitGraph (3322 MREPS) reported much higher numbers on live-journal than AccuGraph (~ 2406 MREPS) in the original articles. However, this is contrary to the absolute numbers we report here as runtime in seconds (Fig. 13a). HitGraph performs worse on all graphs (the numbers in the runtime chart are the factor calculated by dividing the HitGraph runtime by the AccuGraph runtime). Even the simulation inaccuracy of a mean percentage error of 8.997% for WCC measured in Sect. 4.1 cannot change this. This leads us to a first observation that REPS (used as a performance indicator in the original articles) is not a reliable performance measure due to it hiding differences in runtime.

When comparing the two approaches, we notice that AccuGraph needs fewer iterations for WCC than HitGraph (cf. Fig. 13b). AccuGraph converges on a solution quicker because it updates values directly. Due to the two-staged approach of HitGraph, it always works on the values of the past iteration. Lower iteration count is exhibited especially by measurements on high average degree, low-diameter graphs (e. g., slash-dot and orkut). Additionally, AccuGraph shows relatively higher performance for small graphs (e. g., slash-dot and dblp). In this scenario, all vertex value reads besides the partition prefetch are served from low-latency, on-chip BRAM, because there is only one partition. The last two factors for AccuGraphs higher performance are: HitGraph needs more requests to read the edges of the graph *and* the updates, and HitGraph reads 64bit per edge while AccuGraph only reads 32bit

per edge due to the CSR format. We thus expect a performance advantage of at least factor 2 on all measurements which is not achieved by AccuGraph on the rmat-24-16 graph. This is due to the partition skipping optimization of HitGraph (not available for AccuGraph). This leads to our second observation that AccuGraph has a categorical advantage over HitGraph because of its direct application of value changes and compressed graph format.

4.3 Summary – Error Analysis

We saw that our simulation environment is able to reproduce the ground truth performance measurements of the original articles with reasonable error (Sect. 4.1). This is possible for bandwidth-bound algorithms (like HitGraph and AccuGraph) despite the radical hypothesis of disregarding FPGA internals. Especially if relative performance behaviour of approaches is so significantly different (cf. Sect. 4.2), an average error of e. g., 8.997% for WCC is reasonable to make sound relative comparisons. However, we also identified six sources of errors which we discuss in the following. For measurements with insufficiently specified input parameters like start vertices (error source ❸) and edge weights (error source ❹) we see large errors for some graphs. Additionally, we attribute at least some of the error to noise in the measurements. For example, very low runtime measurements like individual iterations of SpMV and PR (error source ❷) can lead to significant noise. We see overestimation of runtime due to missing modelling of pipeline bubbles that slow down request generation or missing modelling of e. g., floating point units that perform complicated calculations (error source ❻). Lastly, there remain two graphs in twitter and youtube for which we cannot explain performance differences based on our simulation but rather attribute these differences to different data sets or different usage of them (error sources ❶ and ❺).

One not easily quantifiable, possible error source (error source ❷) we want to add here is interpretation based on understanding of the original article’s description of their approach. This was e. g., especially necessary for data structures with missing data type specifications. To aid researchers trying to understand the approaches we specified the data types in Tab. 3 and advise to completely specify such parameters in the future to aid reproduction of results.

Without SSSP, we see a low mean error of 14.32%. Thus, for certain use cases, we confirm our hypothesis: *Memory access patterns dominate the overall runtime of graph processing such that disregarding the internal data flow results in a reasonable error of a simulation.* We advise that the simulation should be used in use cases where relative performance behaviour is compared rather than where absolute performance should be estimated. Additionally, if the relative performance behaviour is close for the compared approaches our simulation approach might lead to inaccurate conclusions.

5 Example for Faster Graph Accelerator Engineering

In this section, we illustrate how our approach helps to speed up graph processing accelerator engineering by the example of two enhancements of AccuGraph that we found while analyzing the performance in the previous section. Note that instead of implementing the

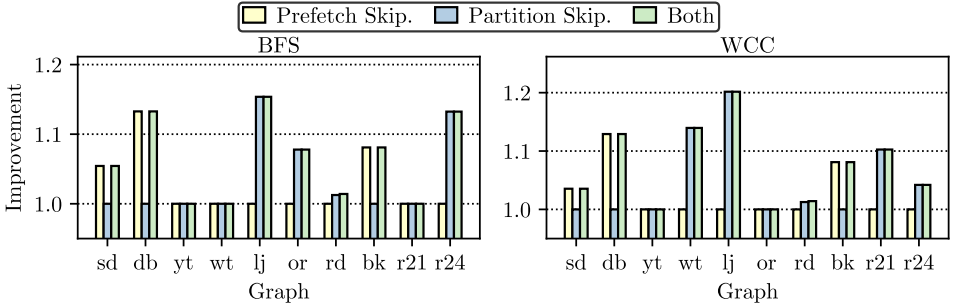


Fig. 14: Runtime improvement of optimizations over baseline

enhancements on the FPGA itself, our simulation approach is used to quickly assess the altered designs for the different data sets as well as potentially different DRAM types, thus reducing the overall engineering time by a form of rapid graph accelerator prototyping.

Enhancement ideas AccuGraph writes all value changes through to off-chip memory and also applies them to BRAM if they are in the current partition. Thus, BRAM and off-chip memory are always in sync. Nevertheless, at the beginning of processing a partition, the value set is prefetched even if the values are already present in BRAM. Thus, the first optimization we propose is prefetch skipping in this case. Especially for the rmat-24-16 graph we also saw the effectiveness of partition skipping with HitGraph (cf. Fig. 13). Thus as a second optimization, we propose adding partition skipping to AccuGraph. Both optimizations can easily be added to AccuGraphs control flow by directly triggering the value and pointer reading producers or completely skipping triggering of execution for certain partitions respectively. For prefetch skipping we compare the currently fetched partition with the next partition to prefetch and skip prefetching if they are the same. For partition skipping we keep track if any value of the vertices of a partition were changed and skip the partition if none changed. The optimizations also work in combination.

Results To prove their effectiveness, we measure the effect of both optimizations for BFS and WCC separately and combined (Fig. 14). For all small graphs with only one partition we see an improvement based on prefetch skipping. Partition skipping is not applicable to those graphs. For some other graphs we see an improvement based on partition skipping. Prefetch skipping only sometimes contributes a small improvement but only when combined with partition skipping. PR as a stationary algorithm is not shown, since no partitions can be skipped by definition. For prefetch skipping there are similar performance improvements on PR compared to BFS and WCC. Overall we see no decrease in performance, suggesting that both optimizations should always be applied.

Note that these insights on the two enhancement ideas were possible in a relatively short amount of time, compared to engineering on an actual FPGA. Developing and testing a complicated FPGA design usually takes weeks, while the implementation of a new graph accelerator approach in our simulation environment takes days or even just hours if the

approach is well understood before. Additionally, the iteration time is much improved. Synthesis runs for compiling hardware description code to FPGA take hours up to a day without many possibilities of incremental synthesis, while a complete compilation of our simulation environment takes 33.5 seconds on a server with the possibility of easily utilizing parameters and incremental compilation. As a downside, the simulation runs longer than a synthesized design on an FPGA would. However, the user is not limited by special hardware that is only available in limited numbers (FPGAs). Many runs can be executed in parallel on one or even multiple servers. Especially for the very fragmented FPGA market, virtualized offers for FPGAs might not be available for specific boards.

6 Related Work

To the best of our knowledge, there are no prior works on only using the off-chip memory requests paired with a DRAM simulator to make graph processing accelerators more comprehensible and performance measurements reproducible and comparable.

Cache miss runtime estimation [MBK02] describe a cost model to approximate query runtimes in relational databases based on cache misses of memory requests. They focus on CPU cache hierarchies which allow much less fine-granular data placement than FPGA memory hierarchies (cf. Sect. 2.2). Additionally, they do not perform simulations of requests but model performance theoretically based on the model parameters of number of cache misses and cache latency not applicable to FPGAs.

Comprehensibility [ZCP15] introduces a DRAM model and simulation for HitGraph. The simulation also generates the sequence of requests, but instead of simulating DRAM runtime, it assumes that every request results in a row buffer hit and models the performance along the cycles needed for processing the data and approximated pipelines stalls. However, they do not show performance numbers generated with this simulation. [Ya19] uses Ramulator as the underlying DRAM simulator for a custom cycle-accurate simulation of the accelerator Graphicionado [Ha16]. However, this incurs very high implementation time.

Reproducibility Regarding reproducibility, there are prior works on ways to report performance results in such a way that it suits the own approach on parallel computing systems [Da95, HB15]. The graph processing accelerator domain seems to suffer from similar problems and lack of widely accepted standards in benchmarking.

Comparability Ramulator [KYM16] was previously used in a work studying the interactions of complex workloads and DRAM types [Gh19]. They uncovered how the internal structure and characteristics of DRAM (DDR3 and DDR4 in our work) relate to performance gains or losses on otherwise fix benchmarks. This may be a future angle to improve graph processing accelerator performance by fitting the DRAM type to the algorithms and data sets. Similarly to our work, [Xu17] raises awareness to lacking comparability in graph processing approaches, but on CPU-based cloud platforms. They find tradeoffs in approaches between different workloads and differently structured graphs.

7 Discussion and Outlook

In this article, we propose a simulation environment for graph processing accelerator approaches based on our hypothesis: *Memory access patterns dominate the overall runtime of graph processing such that disregarding the internal data flow results in a reasonable error of a simulation.* The simulation environment models request flow fed into a DRAM simulator (i. e., Ramulator [KYM16]) and control flow based on data dependencies. We developed a set of memory access abstractions and applied these to FPGA implementations (i. e., HitGraph [Zh19] and AccuGraph [Ya18]) representing the two dominating graph processing approaches (i. e., edge- and vertex-centric).

Even though the simulation environment disregards large parts of the graph processing accelerator, we showed that it is able to reproduce ground truth measurements with a reasonable error for most workloads. In our analysis of the large errors for some workloads we found insufficiencies in benchmark setups and attribute some error to the radical hypothesis of our approach. We further utilized the simulation environment to compare the two approaches on a fixed configuration, revealing insufficiencies in existing performance measurements of graph processing accelerators. Lastly, we show that our simulation approach significantly reduces the iteration time to develop and test graph processing approaches for hardware accelerators by example of two optimizations for AccuGraph that we propose. In addition, our approach allows for deeper inspection with DRAM statistics as well as easy parameter variation without a fixed hardware platform.

In future work, we will extend the approach to an analytical performance model and study the relationship between DRAM types (e. g., HBM, HMC, or LPDDR) and workload types, as well as further graph processing accelerator approaches in more detail. Additionally, there are open questions on how to reduce the relative errors of the simulation environment. This could, e. g., be achieved by studying the simulation environment in more depth on a graph accelerator we implemented and fully control the benchmark setup for.

References

- [Ab19] Abadi, Daniel; Ailamaki, Anastasia; Andersen, David; Bailis, Peter; Balazinska, Magdalena et al.: The Seattle Report on Database Research. *SIGMOD Rec.*, 48(4):44–53, 2019.
- [Be19] Besta, Maciej; Peter, Emanuel; Gerstenberger, Robert; Fischer, Marc; Podstawski, Michal; Barthels, Claude et al.: Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR*, abs/1910.09017, 2019.
- [BRS13] Bacon, David F.; Rabbah, Rodric M.; Shukla, Sunil: FPGA Programming for the Masses. *ACM Queue*, 11(2):40, 2013.
- [Ch12] Chatterjee, Niladrish; Balasubramonian, Rajeev; Shevgoor, Manjunath; Pugsley, Seth; Udipti, Aniruddha; Shafiee, Ali; Sudan, Kshitij; Awasthi, Manu; Chishti, Zeshan: USIMM: the Utah SIMulated Memory Module. University of Utah, Tech. Rep., pp. 1–24, 2012.
- [Da95] Davison, Andrew: Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers. pp. 38–42, 1995.

- [Da17] Dai, Guohao; Huang, Tianhao; Chi, Yuze; Xu, Ningyi; Wang, Yu; Yang, Huazhong: ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In: FPGA. pp. 217–226, 2017.
- [Dr07] Drepper, Ulrich: What Every Programmer Should Know About Memory. Red Hat, Inc, 11, 2007.
- [DRF20] Dann, Jonas; Ritter, Daniel; Fröning, Holger: Non-Relational Databases on FPGAs: Survey, Design Decisions, Challenges. CoRR, abs/2007.07595, 2020.
- [Gh19] Ghose, Saugata; Li, Tianshi; Hajinazar, Nastaran; Cali, Damla Senol; Mutlu, Onur: Demystifying Complex Workload-DRAM Interactions: An Experimental Study. Proc. ACM Meas. Anal. Comput. Syst., 3(3):60:1–60:50, 2019.
- [Ha16] Ham, Tae Jun; Wu, Lisa; Sundaram, Narayanan; Satish, Nadathur; Martonosi, Margaret: Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In: MICRO. pp. 56:1–56:13, 2016.
- [HB15] Hoefler, Torsten; Belli, Roberto: Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In: SC. pp. 73:1–73:12, 2015.
- [KYM16] Kim, Yoongu; Yang, Weikun; Mutlu, Onur: Ramulator: A Fast and Extensible DRAM Simulator. IEEE Comput. Archit. Lett., 15(1):45–49, 2016.
- [LK14] Leskovec, Jure; Krevl, Andrej: , SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>, June 2014.
- [Lu07] Lumsdaine, Andrew; Gregor, Douglas; Hendrickson, Bruce; Berry, Jonathan: Challenges in Parallel Graph Processing. Parallel Processing Letters, 17(01):5–20, 2007.
- [MBK02] Manegold, Stefan; Boncz, Peter A.; Kersten, Martin L.: Generic Database Cost Models for Hierarchical Memory Systems. In: PVLDB. pp. 191–202, 2002.
- [RCJ11] Rosenfeld, Paul; Cooper-Balis, Elliott; Jacob, Bruce: DRAMSim2: A Cycle Accurate Memory System Simulator. IEEE Comput. Archit. Lett., 10(1):16–19, 2011.
- [Xu17] Xu, Chongchong; Zhou, Jinhong; Lu, Yuntao; Sun, Fan; Gong, Lei; Wang, Chao; Li, Xi; Zhou, Xuehai: Evaluation and Trade-offs of Graph Processing for Cloud Services. In: IEEE ICWS. pp. 420–427, 2017.
- [Ya18] Yao, Pengcheng; Zheng, Long; Liao, Xiaofei; Jin, Hai; He, Bingsheng: An Efficient Graph Accelerator with Parallel Data Conflict Management. In: PACT. pp. 8:1–8:12, 2018.
- [Ya19] Yan, Mingyu; Hu, Xing; Li, Shuangchen; Akgun, Itir; Li, Han; Ma, Xin; Deng, Lei; Ye, Xiaochun; Zhang, Zhimin; Fan, Dongrui; Xie, Yuan: Balancing Memory Accesses for Energy-Efficient Graph Analytics Accelerators. In: ISLPED. pp. 1–6, 2019.
- [ZCP15] Zhou, Shijie; Chelmiss, Charalampos; Prasanna, Viktor K.: Optimizing memory performance for FPGA implementation of PageRank. In: ReConfig. pp. 1–6, 2015.
- [Zh19] Zhou, Shijie; Kannan, Rajgopal; Prasanna, Viktor K.; Seetharaman, Guna; Wu, Qing: HitGraph: High-throughput Graph Processing Framework on FPGA. IEEE Trans. Parallel Distrib. Syst., 30(10):2249–2264, 2019.
- [ZL18] Zhang, Jialiang; Li, Jing: Degree-aware Hybrid Graph Traversal on FPGA-HMC Platform. In: FPGA. pp. 229–238, 2018.

A Appendix

Algorithm	Measurement type	Graph						
		berkstan	wikitalk	roadnet	live-journal	twitter	rmat-21	rmat-24
SpMV	Ground truth	0.0032	0.0050	0.0028	0.0362	0.6525	0.0567	0.1435
	Simulation	0.0026	0.0066	0.0027	0.0411	0.8184	0.0484	0.0770
PR	Ground truth	0.0030	0.0045	0.0027	0.0327	0.5904	0.0534	0.1403
	Simulation	0.0026	0.0066	0.0027	0.0411	0.8184	0.0484	0.0770
SSSP	Ground truth	0.7824	0.0255	1.1133	0.5921	5.5768	0.9671	0.9213
	Simulation	1.2554	0.0027	1.3436	0.3872	6.2380	0.0725	0.1111
WCC	Ground truth	1.7690	0.0460	1.4800	0.4130	6.6170	0.4500	1.1080
	Simulation	1.8578	0.0461	1.4526	0.4694	9.4139	0.4653	0.9307

Tab. 5: HitGraph measurements in seconds

Algorithm	Measurement type	Graph					
		slashdot	dblp	youtube	wikitalk	live-journal	orkut
BFS	Ground truth	2.867	2.397	1.899	1.653	3.370	3.638
	Simulation	2.880	2.515	2.530	1.999	2.946	3.192
PR	Ground truth	2.242	1.931	1.560	1.318	1.921	2.587
	Simulation	2.518	1.944	1.978	1.283	1.926	2.920
WCC	Ground truth	2.950	2.468	1.954	1.729	2.407	3.365
	Simulation	2.634	2.183	2.284	1.532	2.254	2.998

Tab. 6: AccuGraph measurements in GREPS

Umbra as a Time Machine: Adding a Versioning Type to SQL

Lukas Karnowski¹ Maximilian E. Schüle² Alfons Kemper³ Thomas Neumann⁴

Abstract: Online encyclopaedias such as Wikipedia rely on incremental edits that change text strings marginally. To support text versioning inside of the Umbra database system, this study presents the implementation of a dedicated data type. This versioning data type is designed for maximal throughput as it stores the latest string as a whole and computes previous ones using backward diffs. Using this data type for Wikipedia articles, we achieve a compression rate of up to 11.9 % and outperform the traditional text data type, when storing each version as one tuple individually, by an order of magnitude.

1 Introduction

Version management of texts is still an important issue due to various use cases. The highlighted example is Wikipedia [Sc17], where people work decentrally on the creation of articles. In order to review their work, version management is mandatory, as it allows administrators to restore any previous version. As even versions of 2001—the founding year of Wikipedia—are accessible, an efficient storage of the data is necessary. Such a data storage should allow fast retrieval of previous versions, new versions to be inserted quickly and consume as little memory as possible.

Temporal databases such *TQuel* [Sn87] or as included in the SQL:2011 standard [KM12] restrict each tuple’s validity to an added time range. In contrast, systems for relational dataset versioning such as *Decibel* [Ma16] lock on a higher granularity to track the history of whole tables. *VQuel* [Ch15], *OrpheusDB* [Hu17] and *LiteTree*⁵ aim at combining SQL [Sc19] and versioning, but do not compress similar text strings. A stand-alone system that includes text compressing is *Forkbase* [Li20] but it is not interoperable with database systems.

```
CREATE TABLE wikidiff (title text, content difftext);
INSERT INTO wikidiff (SELECT 'example', BUILD('first', 'second_version'));
SELECT GET_CURRENT_VERSION(difftext) FROM wikidiff;
```

List. 1: Proposed data type `DiffText` for text versioning.

To measure the potential of compressing text strings, we have benchmarked storing strategies on popular relational database systems using the Wikipedia page edit history. This work

¹ TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, lukas.karnowski@tum.de

² TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, m.schuele@tum.de

³ TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, kemper@in.tum.de

⁴ TU Munich, Chair for Database Systems, Boltzmannstraße 3, 85748 Garching, neumann@in.tum.de

⁵ <https://github.com/aergoio/litetree>

continues the study about versioning in main-memory database systems [SKS+19]: We propose a versioning data type, that can be used as an SQL attribute to store multiple versions of a text within one tuple (see List. 1). The developed data type for the Umbra database system [NF20] is presented in Section 2 by considering the diff algorithm, the memory layout and the implementation of the operations. Section 3 provides an evaluation of the data type's performance. Finally, Section 4 summarises the findings.

2 DiffText Data Type

In this section, we propose a *DiffText* data type to compress multiple versions of a text string as one database attribute within a tuple. The data type is based on the BLOB or TEXT data type that is available in many database systems to store byte sequences of any length. It thus inherits their properties with regard to the memory layout. This includes flexible size within a tuple to be enlarged as required, which is necessary when adding new versions. The data type is used as an SQL attribute: its values are overwritten on updates and copied for each occurrence as a column. This section presents the used algorithm for compressing strings, the data type's memory layout and necessary operations to retrieve versions out of a tuple.

2.1 Delta-Compression Algorithm

The DiffText data type applies delta compression to multiple versions of a string. It relies on difference-based versioning as it stores at least the latest version as a snapshot and restores the remaining ones using relative changes to the current version (*backward diffs*).

The idea is to access each byte of both versions only once. A function `find_diff()` determines the first and the last differing byte between two consecutive versions. First, both texts were compared from the beginning until the first differing byte has been found. The process is then repeated from the end of the texts. All bytes between these two boundaries found are called a *patch* and are part of the resulting diff even if they have bytes in common.

Example: The first step of the call `find_diff(aacbb, addbb)` terminates after the second byte ($a \neq d$). The second step terminates after the third character from the back ($c \neq d$). The resulting diff is the string `ddd`, called *patch*, with the additional information that the second and third characters in the first text must be replaced. A complete diff thus consists of three parts, *patch*, *start* and *end*. The interval at which the patch must be applied is called `patchStart` and `patchEnd`.

When more than two versions exist, an order must be defined in which direction the patches will be applied. We decide in favour of *backward diffs*: The most current version is always available as a complete text, whereas older versions are stored as the difference to the version that was inserted afterwards.

Example: Assuming a newer version T_2 replaces the current one T_1 . Having two similar text strings, the function `find_diff()` calculates the diff $D_{T_2 \rightarrow T_1} =: D_1$, so that T_1 can be reconstructed out of T_2 and D_1 . The entire text of T_1 is then discarded and replaced by D_1 . If another version T_3 is added, the diff $D_{T_3 \rightarrow T_2} =: D_2$ will be calculated and saved to replace T_2 . If we want to reconstruct T_1 , we will first apply D_2 to get T_2 and then apply D_1 to get T_1 .

This process creates a chain of diffs that must be applied to restore older versions. Specifically, the number of involved patches increases with the number of inserted versions. For this reason, it is advisable to periodically save the complete version instead of calculating a diff. This allows constant access times in $O(1)$ to any version. Assuming that every third version should be complete and two additional versions T_4, T_5 are inserted, the chain of diffs would look like in Figure 1. Only two versions are complete and the remaining ones can be restored using diffs.

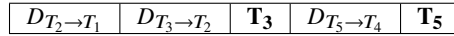


Fig. 1: Chain of diffs, with every third version as a complete snapshot (bold).

2.2 Memory Layout

To enable efficient operations later on, all versions of a text string are stored as one object. This leads us to the memory layout, which corresponds to the output of the presented algorithm out of patches and corresponding ranges. The actual patch is saved separately from the start and end of the diff. Figure 2 shows the schematic representation of the memory layout of the DiffText data type and Figure 3 the associated code.

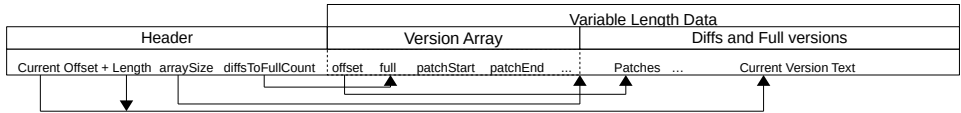


Fig. 2: Structure of a DiffText tuple.

```

struct DiffTextRepresentation {
    uint32_t currentOffset;    // Offset of current version in data section
    uint32_t currentLength;    // Length of current version
    uint32_t arraySize;        // The size of the version pointer's array
    uint16_t diffsToFullCount; // Counter of diffs until next full version
    struct {
        uint32_t offset;      // Array of pairs, pointing into the data section
        bool full;            // Offset of version in data section
        uint32_t patchStart;   // Is this a full version?
        uint32_t patchEnd;    // Start of patch
    } versionPointers[];      // End of patch
    // Data section follows this struct immediately
};

```

Fig. 3: Source-code of the DiffText representation.

The layout starts with a header that indicates the size of the subsequent area. This variable-sized area contains all versions as diffs and is further divided into two parts: The first part contains a version array out of an offset, a flag `full`, and a range (`patchStart`, `patchEnd`). `patchStart` and `patchEnd` indicate the position that need to be changed in order to restore the previous version. The offset acts as a pointer to the last area in which the associated patch is located. Consequently, the last memory section is the concatenation of all patches and complete versions from which any version can be restored.

Since the latest version is always stored as a complete snapshot, the header contains an offset to the latest version in the data area in order to accelerate its access. The header also contains the current number of diffs that must be applied to restore a version (`diffsToFullCount`). Its value is incremented when a new version has been added. After reaching a predefined number, instead of calculating a patch, a complete snapshot will be saved, as presented in Section 2.1. This method ensures that each version can be extracted in $O(1)$. As the text's length is not stored, the tag `full` in the version array indicates whether the corresponding version has been stored as a complete snapshot instead of a patch.

For the offsets in the data area, 32 bit numbers have been used as Umbra's text-based data types are limited to 2^{32} bytes. This restrains the `DiffText` data type as all versions concatenated may not exceed a maximum size of 4 GiB. 16 bit was chosen for `diffsToFullVersion`, to avoid diff chains longer than 65536 as the runtime increases linearly with the number of patches.

Furthermore, only the offset is saved and the length of the patch is omitted. This is possible as the offset of the subsequent diff determines the end of the previous one. An exception is made for the current version, whose length is saved for fast retrieval.

2.3 Example for a `DiffText` object

For a better understanding of the memory layout, this section demonstrates the construction of a `DiffText` object by the following example: The initial version "*First*" will be changed to "*First Version*" by adding "*Version*". Then the current version is set to "*Second Version*". The resulting `DiffText` objects are listed in Figure 4.

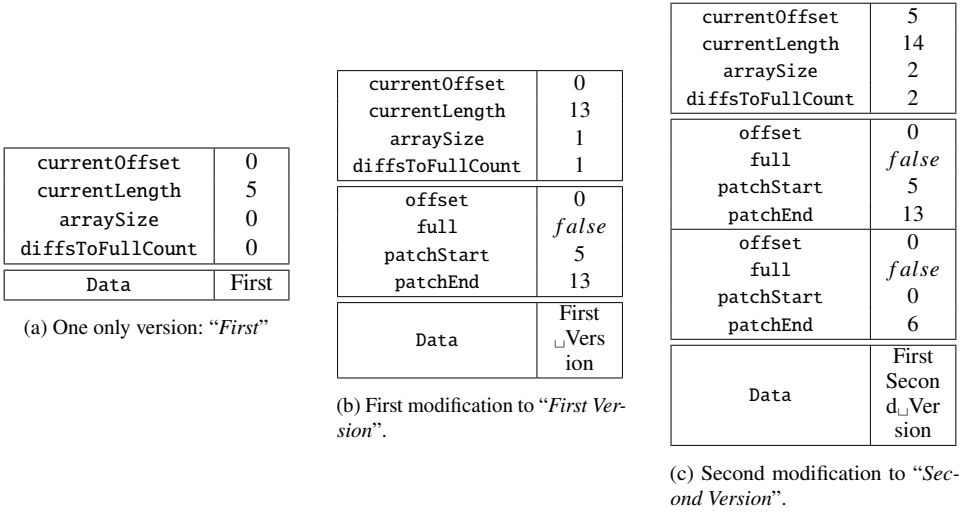


Fig. 4: States of a DiffText object when updating its entry with the following versions: (a) "First", (b) "First Version" and (c) "Second Version". The current version is reconstructed out of the text string in Data using currentOffset and currentLength.

Figure 4a shows the initial state with only one version. The version array is empty (arraySize = 0) and the only content in the variable-sized memory area is the current version "First".

After updating the entry, Figure 4b shows the second state with the version array containing one entry. Since backward diffs are used, this entry contains information on how to restore the previous version "First Version" from the current version "First". In this case, the content has to be cut off after "First". Accordingly, the coded patch in the version is a character string with a length of 0 (offset = 0). Since no length is stored in the version array, the length is implicitly calculated from the start of the subsequent version. In this case the following version is the current one, which is why the field currentOffset is considered. The length of the patch is therefore currentOffset-versionPointers[0].offset=0. The fields patchStart and patchEnd indicate at which point the patch must be inserted: In this case, the interval [5, 13) corresponds to the added character string "_Version".

Figure 4c depicts the final state after inserting "Second Version". The version array now contains two entries: the first entry remains unchanged, whereas the second specifies how to restore "First Version" out of "Second Version". This is done by replacing the word "Second" with "First", so the patch must contain the latter character string. The interval [0, 5) results from the offset entry in the array and currentOffset in the header, i.e. the first five characters in the data area ("First"). This patch is inserted in-between patchStart and patchEnd in the area [0, 6) of the current version, which corresponds to the already mentioned replacement of the first word.

Also of interest is the field `diffsToFullCount`, which is equal to `arraySize` in our example. If more versions are inserted and `diffsToFullCount` reaches a predefined threshold value, a complete version will be saved, which is indicated by the tag `full` in the version array. `diffsToFullCount` is then reset to 0 and the process starts again.

2.4 Implementation of the Corresponding Operations

Since the data type was developed in Umbra, which currently does not support UPDATE operations, a copy of the previous state must be created for each operation. The data type supports the following functions:

- $\text{BUILD}(T_1, \dots, T_N)$ creates a DiffText object from a set of N versions. T_1 corresponds to the oldest version and T_N to the latest one. This can be used for recovery operations, for example, when creating backups out of bare text strings.
- $\text{APPEND}(D, T_1, \dots, T_N)$ is a generalisation of the BUILD operation. It expects a DiffText object D , to which the versions $T_{1..N}$ are appended.
- $\text{SET_CURRENT_VERSION}(D, T)$ is a specialisation of APPEND, as it modifies a single version only, the standard operation for adding a new version.
- $\text{GET_VERSION_BY_ID}(D, N)$ extracts version N from the given DiffText object. SQL is typically indexed starting with 1, with lower numbers indicating older versions and higher numbers corresponding to newer versions.
- $\text{GET_CURRENT_VERSION}(D)$ returns the latest version. If the data type contains M versions in total, it is equivalent to $\text{GET_VERSION_BY_ID}(D, M)$. For performance reasons, a separate and optimised operation is offered to retrieve the latest version. The structure of the DiffText data type is designed to extract the latest version as quickly as possible. This will be discussed later in more detail.

In addition, $\text{EXPAND}(D, M, N)$ is a unary database operator that extracts the versions within the interval $[M, N]$ out of a single DiffText object. It expects a relation with a DiffText column as input and returns $N - M + 1$ output tuples per input tuple. For performance reasons, newer versions appear first (the output order is T_N, T_{N-1}, \dots, T_M).

This subsection presents the implementation of the previously presented operations for the DiffText data type.

2.4.1 Accessing Versions

Accessing an arbitrary version demands for the complete reconstructed text string. This is trivial for $\text{GET_CURRENT_VERSION}$, which is stored as a snapshot. In addition, its access

does not require to query the version array, only the offset and the length are read from the header. Furthermore, instead of allocating memory for the returned string, a view to the substring containing the snapshot is sufficient as return value.

The same optimisation will apply if the version requested by `GET_VERSION_BY_ID` is available as a snapshot (`full = true` in Figure 3). If this is not the case, a new buffer must be created for the return string. For performance reasons, the buffer size must be determined in advance. This is not trivial, since any diffs in-between might increase, decrease or leave the length of the resulting text unchanged. For this reason, `GET_VERSION_BY_ID` consists of 3 steps:

1. *Finding the next complete version.* This iterates from the requested version upwards through the version array until a complete snapshot is found. This could also be the latest version, this special case must be considered, since the most current version is not contained in the version array.
2. *Calculating the buffer size.* This requires again an iteration but in reverse order. In each step, the `patchStart` and `patchEnd` fields are used to calculate the resulting buffer size. The required buffer size is the maximum of all sizes found during all iterations.
3. *Applying patches.* In the last step, the version array is iterated downwards again and the corresponding diff is applied in each step. After this process, the requested version is available in the allocated buffer and ready to be returned.

This explains the separation into `patchStart/patchEnd` information and the actual patches within the memory layout: The first two steps do not require the actual patch, but only the meta information of each diff. This ensures optimal cache utilisation.

A further optimisation applies to `EXPAND`: Instead of iteratively calling `GET_VERSION_BY_ID` for each requested version, the patch is applied incrementally, starting with the last requested version. The implementation first calls `GET_VERSION_BY_ID` for the last requested version and then uses a function `getPreviousVersion(D, T)` to determine the predecessors. This implies that the order of the versions of the `EXPAND` operator is exactly counter-intuitive: starting with newer and ending with older versions. For performance reasons, however, this sequence is advantageous because only one step in the version array has to be carried out for each tuple output.

2.4.2 Creating a `DiffText` object

The trivial case when creating a `DiffText` object is with exactly one existing version. For this, the `currentLength` of the `DiffTextRepresentation` is set to the length of the single version. The remaining fields are initialised with 0, the version array is empty and the variable data area contains the current version only.

If more than one version exists, the resulting object will hold all information for their restoration. For this purpose, the diff is formed between two adjacent text strings by iterating once over all bytes and forming the patch between the current and the subsequent version. If the buffer already contains the text string to be inserted, no patches will be copied, but the corresponding offsets have to be saved. After all patches have been created, the texts are iterated again and the part relevant for the diff is copied into the data section of the newly created DiffText object. Thus BUILD consists of two phases (1) *Calculating the diffs* and (2) *copying the patches to the final buffer*.

APPEND is a generalisation of BUILD, because in addition to the new versions, an existing DiffText object is specified, to which the versions are appended. Apart from this, APPEND does not differ to BUILD, why it will not be discussed in more detail. The same applies to SET_CURRENT_VERSION, the specialisation of APPEND, which reuses the two phases mentioned above.

3 Evaluation

This section discusses the performance of the implemented DiffText data type. The Wikipedia dumps from 09/01/2019 were used as test data, specifically pages 971896 to 972009. The measurements have been conducted on an Ubuntu 18.04 LTS server with an Intel Xeon CPU E5-2660 v2 processor with 2.20 GHz (20 cores) and 256 GiB DDR4 RAM.

The full dump has an uncompressed size of 119.9 MiB. First we evaluate the memory consumption after all available versions have been inserted. We add all versions of all pages in a DiffText object to better estimate the memory consumption. The result is shown in Figure 6. Instead of a patch, a complete snapshot will be stored every 50th version. This threshold, which restricts the chain length of patches, is referred to as X in the following.

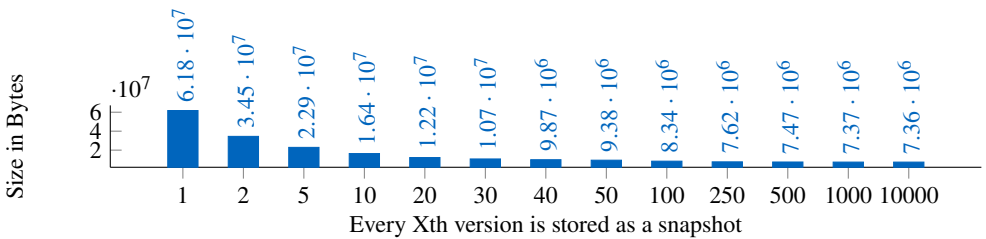


Fig. 5: Memory consumption depending on the frequency of stored snapshots.

The full size of the DiffText object is 8.9 MiB, which is a reduction down to 15.2 % of the original size. Figure 5 shows the total size of the DiffText object depending on the maximum chain length. Once a value of $X = 20$ has been exceeded, the memory consumption does not improve significantly the longer the chains become.

The best improvement achieve a chain length of $X = 10000$ with a reduction to 11.9 % of the original size. Compared to a value of $X = 50$, this means an improvement of only 3.3 percentage points condoning slower access to older versions. In [SKS+19] we achieved a compression to 5 %, which could not be reproduced in this work as the used Wikipedia dump includes less versions per article.

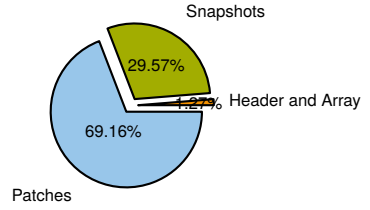


Fig. 6: Memory consumption with a complete snapshot every 50th version.

Let us now consider the runtime of the operations. A comparison with the normal TEXT data type is made by inserting the same versions of a text into a table with TEXT data as well as into a DiffText object. All revisions of one article are stored as one single DiffText tuple, while each snapshot is stored individually as a tuple. The comparison is therefore not representative as it compares different functions of the database system with one another. Nevertheless, the same amount of information is stored in both cases and a tenth of the memory is consumed in the case of the diff approach.



Fig. 7: Comparison: Storing each version as a single snapshot or in one DiffText object.

The following query inserts data into DiffText objects and retrieves text strings out of them:

```
INSERT INTO t (text) VALUES (BUILD(T1, ..., TN)); SELECT EXPAND(text, 1, N) from t;
```

List. 2: Benchmark queries using the DiffText data type.

The snapshots of all texts are inserted into the database as follows and then queried again:

```
INSERT INTO t (rev_id, text) VALUES (1, T1), ..., (N, TN); SELECT text from t;
```

List. 3: Benchmark queries using one tuple for each version.

Figure 7 compares the cumulative compilation and execution times of both approaches. The diff approach performs better than the snapshot approach in all metrics. The snapshot approach creates a tuple for each version and requires significantly more operations to insert the content.

4 Conclusion

In this work an implementation of a diff-based data type was presented, which is required for use cases like Wikipedia. New versions are created regularly, although older texts must still be accessible. The data type presented is implemented for the Umbra database system and is based on the normal text data type. The memory layout is designed for cache efficiency and consists of a header, a version array and the data area with patches and complete versions. The diff algorithm used is simple and can create diffs with just a single pass over the text.

The data type achieves a compression rate of up to 11.9 % of the original size for Wikipedia articles and is faster than the direct comparison with normal texts in both compilation and execution time. No other database system offers a similar data type so far, and research in this area is rather limited. Possible future optimisations for the data type include a larger storage capacity, storing older versions on background memory and a diff algorithm with stronger compression.

References

- [Ch15] Chavan, A. et al.: Towards a Unified Query Language for Provenance and Versioning. In: TaPP. USENIX Association, 2015.
- [Hu17] Huang, S. et al.: OrpheusDB: Bolt-on Versioning for Relational Databases. Proc. VLDB Endow. 10/10, pp. 1130–1141, 2017.
- [KM12] Kulkarni, K. G.; Michels, J.-E.: Temporal features in SQL: 2011. SIGMOD Rec. 41/3, pp. 34–43, 2012.
- [Li20] Lin, Q. et al.: ForkBase: Immutable, Tamper-evident Storage Substrate for Branchable Applications. In: ICDE. IEEE, pp. 1718–1721, 2020.
- [Ma16] Maddox, M. et al.: Decibel: The Relational Dataset Branching System. Proc. VLDB Endow. 9/9, pp. 624–635, 2016.
- [NF20] Neumann, T.; Freitag, M. J.: Umbra: A Disk-Based System with In-Memory Performance. In: CIDR. www.cidrdb.org, 2020.
- [Sc17] Schüle, M. E. et al.: Monopedia: Staying Single is Good Enough - The HyPer Way for Web Scale Applications. Proc. VLDB Endow. 10/12, pp. 1921–1924, 2017.
- [Sc19] Schüle, M. E. et al.: The Power of SQL Lambda Functions. In: EDBT. Open-Proceedings.org, pp. 534–537, 2019.
- [SKS+19] Schüle, M. E.; Karnowski, L.; Schmeißer, J., et al.: Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB. In: SSDBM. ACM, pp. 169–180, 2019.
- [Sn87] Snodgrass, R. T.: The Temporal Query Language TQuel. ACM Trans. Database Syst. 12/2, pp. 247–298, 1987.

ML & Data Science

Aggregate-based Training Phase for ML-based Cardinality Estimation

Lucas Woltmann¹, Claudio Hartmann¹, Dirk Habich¹, Wolfgang Lehner¹



Abstract: Cardinality estimation is a fundamental task in database query processing and optimization. As shown in recent papers, machine learning (ML)-based approaches may deliver more accurate cardinality estimations than traditional approaches. However, a lot of training queries have to be executed during the *model training phase* to learn a data-dependent ML model making it very time-consuming. Many of those training or example queries use the same base data, have the same query structure, and only differ in their selective predicates. To speed up the model training phase, our core idea is to determine a *predicate-independent pre-aggregation* of the base data and to execute the example queries over this pre-aggregated data. Based on this idea, we present a specific *aggregate-based training phase* for ML-based cardinality estimation approaches in this paper. As we are going to show with different workloads in our evaluation, we are able to achieve an average speedup of 63 with our *aggregate-based training phase* and thus outperform indexes.

Keywords: cardinality estimation; machine learning; database support; pre-aggregation

1 Introduction

Due to skew and correlation in data managed by database systems (DBMS), query optimization is still an important challenge. The main task of query optimization is to determine an efficient execution plan for every SQL query, whereby most of the optimization techniques are cost-based [Le15]. For these techniques, cardinality estimation has a prominent position with the task to approximate the number of returned tuples for every query operator within a query execution plan [HN17, Le15, MNS09, YW79]. Based on these estimations, various decisions are made by different optimization techniques such as choosing (i) the right join order [FM11], (ii) the right physical operator variant [Ro15], (iii) the best-fitting compression scheme [Da19], or (iv) the optimal operator placement within heterogeneous hardware [KHL17]. However, to make good decisions in all cases, it is important to have cardinality estimations with high accuracy.

As shown in recent papers [Ki19b, Li15], including our own work [Wo19b], machine learning-based cardinality estimation approaches are able to meet higher accuracy requirements, especially for highly correlated data. While traditional approaches such as histogram-based and frequent values methods assume data independence for their estimation [Le15], ML-based approaches assume that a sufficiently deep neural network can

¹ Technische Universität Dresden, Database Systems Group, 01062 Dresden, Germany,
firstname.lastname@tu-dresden.de

model the very complex data dependencies and correlations [Ki19b]. For this reason, ML-based cardinality estimation approaches may thus give much more accurate estimations as clearly demonstrated in [Ki19b, Li15, Wo19b]. However, the main drawback of these ML-based techniques compared to traditional approaches is the high construction cost of the *data-dependent* ML-models based on the underlying supervised learning approach. During the so-called *training phase*, the task of *supervised learning* is to train a model, or more specifically learn a function, that maps input to an output based on example (input, output) pairs. Thus, in the case of cardinality estimation, many pairs consisting of (query, output-cardinality) are required during the *training phase*. To determine the correct output-cardinalities, the queries have to be executed [Ki19b, Wo19b], whereby the execution of those example queries can be very time consuming, especially for databases with many tables, many columns, and millions or billions of tuples resulting in a heavy load on the database system.

Core Contribution. To overcome these shortcomings, we propose a novel training phase *based on pre-aggregated data* for ML-based cardinality estimation approaches. Usually, as described in [Ki19b, Wo19b], every example query is (i) rewritten with a count aggregate to retrieve the correct output-cardinality and (ii) executed individually. However, many of those example queries use the same base data, have the same query structure, and only differ in their selective predicates. To optimize the query execution, our core idea is to provide a *predicate-independent pre-aggregation* of the base data and to execute the example queries over this pre-aggregated data. Consequently, the set of similar example queries has to read and process less data because the *pre-aggregation* is a compact representation of the base data. To realize this *pre-aggregation*, the most common solution in DBMS is to create a *data cube* for storing and computing aggregate information [Gr96]. However, this *pre-aggregation* is only beneficial if the execution of the example queries on the data cube plus the time for creating the data cube is faster than the execution of the example queries over the base data. As we are going to show with different workloads of example queries in our evaluation, we are able to achieve an average speedup of 63. We also compare our approach to standard query optimization with index structures on the base data and show their limited benefit for this use case.

Contributions in Detail and Outline. Our *aggregate-based training phase* consists of two phases: (i) creation of a set of meaningful *pre-aggregated data sets* using data cubes and (ii) rewrite and execute the example queries on the corresponding data cubes or the base data. In detail, the contributions in this paper are:

1. We start with a general overview of ML processes in DBMS in Section 2. In particular, we detail cardinality estimation as a case study for ML in DBMS. We introduce *global* and *local models* as two representatives for ML-based cardinality estimation approaches. Primarily, we show their properties in terms of example workload complexity and conclude the need for optimization of such workloads.
2. Based on this discussion, we introduce our general solution approach of an *aggregated-based training phase* by *pre-aggregating* the base data using the *data cube* concept and

executing the example queries over this pre-aggregated data. Moreover, we introduce a *benefit criterion* to decide whether the *pre-aggregation* is beneficial or not.

3. In Section 4, we present our *aggregate-based training phase* for ML-based cardinality estimation approaches in detail. Our approach consists of two components: *Analyzer* and *Rewrite*. While the main task of the *Analyzer* component is to find and build all beneficial data cubes, the *Rewrite* component is responsible for rewriting the example queries to the constructed data cubes if possible.
4. Then, we present experimental evaluation results for four different workloads for the training phase of ML-based cardinality estimation in Section 5. The workloads are derived from different ML-based cardinality estimation approaches [Ki19b, Wo19b] on the IMDB data set [IM17]. Moreover, we compare our approach with the optimization using index structures.

Finally, we conclude the paper with related work in Section 6 before concluding in Section 7.

2 Machine Learning Models for DBMS

In this section, we start with a brief description of the general process of *machine learning (ML)* in the context of DBMS. Moreover, we discuss ML-based cardinality estimation for DBMS as an important case study and revisit two ML-based approaches solving this specific challenge. Finally, we analyze the specific query workloads for the training phases and clearly state the need for optimized database support.

2.1 Machine Learning Support for DBMS

Most ML-supported techniques for DBMS are supervised learning problems. In this category, there are amongst others: cardinality estimation [Ki19b, Li15, Wo19b], plan cost modeling [MP19, SL19], and indexing [Kr18]. The proposed ML solutions for those highly relevant DBMS problems have a general process in common as shown in Figure 1. This process is usually split into two parts: *forward pass* and *training phase*.

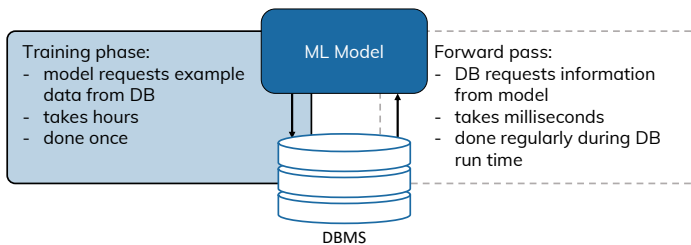


Fig. 1: The general process of supervised ML in DBMS.

Forward pass: This pass consists of the application of the model triggered by a request of the DBMS to the ML model. Each request pulls some specific information from the model such as an estimated cardinality or an index position [Ki19b, Kr18, Li15, Wo19b]. The execution time of each forward pass request is normally in the range of milliseconds. This is advantageous because forward passes occur often and regularly during the run time of the DBMS [Ki19b, Kr18, Li15, Wo19b].

Training phase: To enable the forward pass, a training phase is necessary to construct the required ML model, whereby the challenge for the model lies in the generalization from example data [Ki19b, Kr18, Li15, Wo19b]. Therefore, the model usually requests a lot of diverse labeled example data—pairs of (input, output)—from the DBMS to learn the rules of the underlying problem. Even though the training is performed once, its run time may take hours. This is mainly caused by the generation and execution of a large number of queries against the DBMS to determine the correct cardinalities.

As a consequence, the *training phase* of ML models to support DBMS usually generates a spike high load on the DBMS. Compared to the *forward pass*, the training is significantly more expensive from a database perspective. Therefore, the training phase is a good candidate for optimization to reduce (i) the time for the training phase and (ii) the spike load on the DBMS. Thus, database support or optimization of the training phase is a novel and interesting research field leading to an increased applicability of ML support for DBMS.

2.2 Case Study: Cardinality Estimation

As already mentioned in the introduction, we restrict our focus to the ML-based cardinality estimation use case [Ki19b, Li15, Wo19b]. In this setting, each *forward pass* requests an estimated cardinality for a given query from the ML model. In the *training phase*, the ML cardinality estimator model requires example queries as example data from the DB where the queries are labeled with their true cardinality resulting in pairs of (query, cardinality). These cardinalities are retrieved from the DB by executing the queries enhanced with a count aggregate. Two major approaches for user-workload-independent cardinality estimation with ML models have been proposed in recent years: *global* and *local models*.

Global Model Approach

A *global model* is trained on the *complete database schema* as the underlying model context [Ki19b]. It is effective in covering correlations in attributes for high-quality estimates [Ki19b]. In Figure 2, this is depicted by a single model stated and mapped to the complete schema. Global models have downsides like (i) the complexity of the ML model and (ii) the very expensive training phase. Both disadvantages arise for the following reason: the single ML model handles all attributes and joins in the same way leading to a huge problem space. This huge problem space is directly translated to the model complexity as

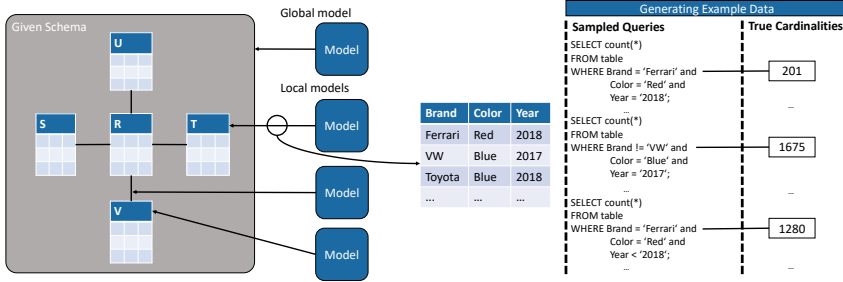


Fig. 2: Overview of ML-based cardinality estimation approaches.

well as to a high number of example queries to cover all predicates and joins over the whole schema as shown [Ki19b].

Local Model Approach

To overcome the shortcomings of the global model approach, the concept of *local models* has been introduced [Wo19b]. Local models are ML models which only cover a certain sub-part of the complete database schema as their model context. This can be a base table or any n-way join. Again, Figure 2 details several local models each covering a different part of the schema. As each of them focuses on a part of the schema, there are many advantages compared to global models. Firstly, local models produce estimates of the same quality as global models [Wo19b]. Secondly, their model complexity is much smaller, because they cover a smaller problem space in different combinations of predicates and joins. The lower complexity stems from a more focused or localized problem solving. A local model has to generalize a smaller problem than the global, i.e. the cardinality estimate of a sub-part of a schema and not the whole schema at once. The lower complexity leads to faster example query sampling and training because the easier problem requires fewer queries during training. A major disadvantage of local models is the high number of models needed to cover all objects touched by a query within the forward pass. Therefore, a separate local ML model needs to be constructed for each part of the schema. Additional queries need to be generated because every local model requests the same amount of example queries. However, these queries are less complex because there are fewer combinations of predicates and tables in a local context.

2.3 Training Phase Workload Analysis

Fundamentally, the global as well as the local ML-based cardinality estimation approach use the same method to sample example queries for the *training phase*. This procedure is shown on the right side of Figure 2, where a local model is trained on an example table

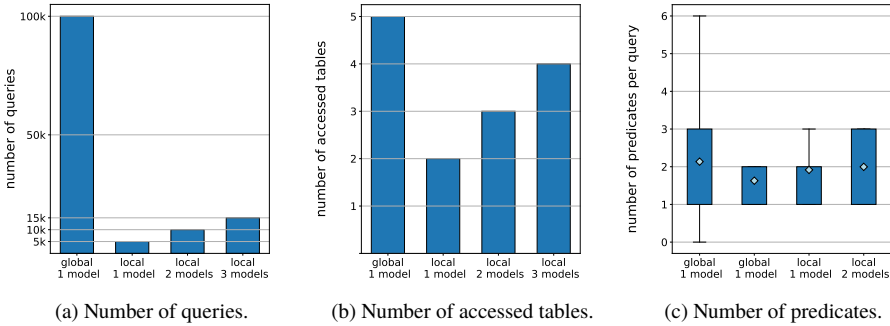


Fig. 3: Analysis of ML-workload complexity for 1 global and 1 to 3 local models for the IMDB.

consisting of three attributes Brand, Color, and Year. To train the local *data-dependent* ML-model, a collection of count queries with different predicate combinations over the table is generated. In our example, the predicates are specified over the three attributes using different operators \neq , $<$, \leq , $=$, $>$, \geq and different predicate values. Thus, all queries have the same structure but differ in their predicates to cover every aspect of the data properties in the underlying table. An example query workload to train a global model would look similar. However, the different contexts for global and local models have an impact on the number and the complexity of the example queries. In general, the query complexity is given by the combinations of joined tables and predicates in a query. The larger the model context, the more complex the example queries. Thus, global models have workloads with a higher number of variations for predicates and joins because they cover the whole schema. Local models are trained with workloads with lesser variation [Wo19b].

To better understand the query workload complexity for the training phase, we analyzed the workloads published by the authors of the global [Ki19a] and local approach [Wo19a]. In both cases, the authors used the IMDB database [IM17] for their evaluation, because this database contains many correlations and is thus very challenging for cardinality estimation. Our analysis results are summarized in Figure 3 and 4.

For a global model and an increasing number of local models, Figure 3a shows the workload complexity in terms of numbers of example queries used per workload. While the global model requires up to 100,000 example queries for the IMDB database [Ki19b], the local model only requires 5,000 example queries per local model [Wo19b] to determine a *stable data-dependent* ML model. In general, the number of example queries is much higher for a global model, but the number of example queries also increases with the number of local models. Thus, we can afford more local models before their collective query count exceeds the number of queries for the global model.

Figure 3b specifies the workload size in terms of data access through the number of joined tables per workload. Similar to the previous figure, the global model has the highest complexity because it requires example queries over more tables to cover the whole schema

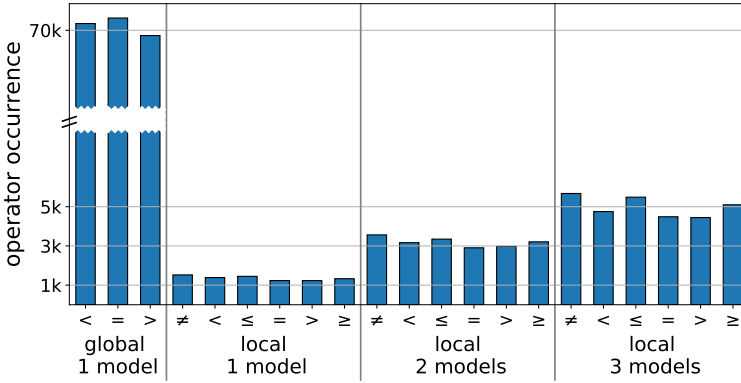


Fig. 4: Predicate operator occurrences for IMDB.

at once. Each local model covers only a limited part of the schema and therefore queries fewer tables per model. Thereby, the complexity of accessed data for local models increases with the number of models.

Another important aspect to describe the workload complexity is the number of predicates per query as shown in Figure 3c. Here, the global model workload has a much larger spread over the number of predicates. The local models detail a more focused distribution with little variation. Again, the local model workload does not require the amount of alternation in predicates of a global model workload because it covers a smaller fragment of the schema. Additionally, Figure 4 gives an overview of the distribution of occurrences of all predicate operators in the workloads as a box plot with mean values. The global model workload only uses the operators $<$, $=$, $>$, whereas the local model workloads use the full set of operators \neq , $<$, \leq , $=$, $>$, \geq . As described by both authors, the predicate operators in each example query are sampled from a uniform distribution [Ki19b, Wo19b]. The slight variation between the operators per workload is due to the fact that both approaches filter 0-tuple queries which do not occur uniformly.

3 Training on Pre-Aggregated Data

As discussed above, the global as well as the local ML-based approach for cardinality estimation generates many example queries with a count aggregate function during the *training phase*. Depending on the model context, there is a small variance in the number of accessed tables, but there is a high variance for predicates in terms of (i) number of predicates, (ii) used predicate operators, and (iii) predicate values in general. So, many queries work on the same data but look at different aspects. Executing such workloads in a naïve way, i.e. executing each example query individually on large base data, is very expensive and generates a high spike load on the database system. The utilization of index

structures for an optimized execution in database systems appears to be an ideal technique at a first glance. However, their benefit is limited as we will show in our evaluation. The same can be said about materialized views. We omitted their evaluation because the experiment did not finish within 30 days.

To tackle this problem more systematically, our core idea is to *pre-aggregate* the base data for different predicate combinations and to reuse this *pre-aggregated* data for several example queries. In general, aggregates compress the data by summarizing information and reducing redundancy. This lessens the amount of data to be scanned by each example query because the aggregates can be smaller than the original data. The aggregate pre-calculates information with the result that the workload queries need to scan less data during execution. Therefore, it is important that the construction of the aggregate does not take longer than the reduction of the workload execution time.

3.1 Grouping Sets as Pre-aggregates

It might sound expensive to aggregate all possible combinations of predicates. However, DBMS already offer substantial supportive data structures for this kind of aggregation. The basic idea of such grouping comes from *Online Analytical Processing* (OLAP) workloads. These aggregate-heavy workloads spawned the idea of pre-aggregating information in *data cubes* [Gr96] helping to reduce the execution time of OLAP queries by collecting and compressing the large amount of data accessed into an aggregate. The concept of data cubes is well-known from data warehouses for storing and computing aggregate information and almost all database systems are offering efficient support for data cube operations [Ag96, Gr96, HRU96, Sh96, ZDN97].

Each attribute of a table or join generates a dimension in the data cube and the distinct attribute values are the dimension values. The cells of a data cube are called facts and contain the aggregate for a particular combination of attribute values. To instantiate the concept of a data cube in a DB, there are different *cube operators*. Usually, these are CUBE, ROLLUP, and GROUPING SET. The CUBE operator instantiates aggregates for the power set of combinations of attribute values. The ROLLUP operator builds the linear hierarchy of

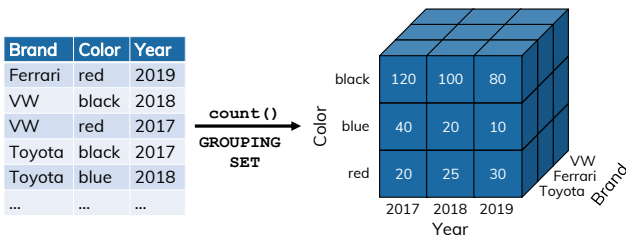


Fig. 5: Aggregating information with grouping sets.

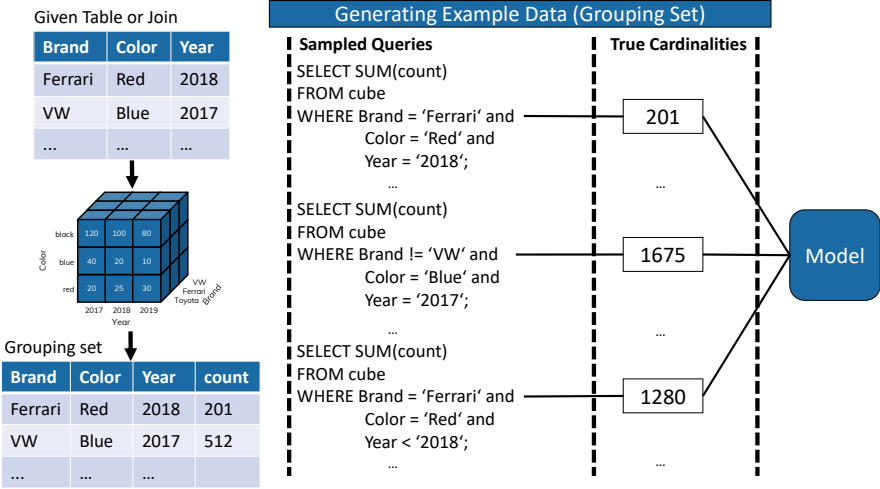


Fig. 6: Illustration of database-supported training phase based on pre-aggregated data/grouping sets.

attribute combinations. The GROUPING SET operator only constructs combinations with all attributes. This characteristic of a grouping set is the major advantage for our use case. With the grouping set aggregation, we compress the original data and avoid the calculation of unnecessary attribute combinations. Figure 5 details an example of a grouping set for a count aggregate over discrete data. The example data has a multidimensional structure after aggregating where each dimension is a property of a car. The cells of the grouping set are filled with the aggregate value, i.e. the count of cars with a particular set of properties.

Given the grouping set data structure, we adapt the generation of example queries from Figure 2. By introducing the data cube, we add an intermediate step before executing the workload. This step constructs a data cube, i.e. grouping set, and rewrites the workload to fit the grouping set. Figure 6 details the construction and the rewrite of queries for the sample data. On the left side, the construction builds a table matching the multidimensional character of the grouping set. Due to this new table layout, the rewrite must include a different aggregate function as shown in Figure 6. For a count aggregate, the corresponding function is a sum over the pre-aggregated count. Last, the rewritten workload is executed and retrieves the output-cardinalities. After the sampling of example queries, the queries and cardinalities are fed to the ML model in the same way as in the original process. This is an advantage of our approach because it does not interfere with other parts of the training process. Therefore, it is independent of the type of ML model and can be applied to a multitude of supervised learning problems.

Even though our approach is independent of the ML model, it is not independent of the data. In general, grouping sets are only beneficial if the aggregate is smaller than the original data. A negative example would be an aggregate over several key columns. Here, the number

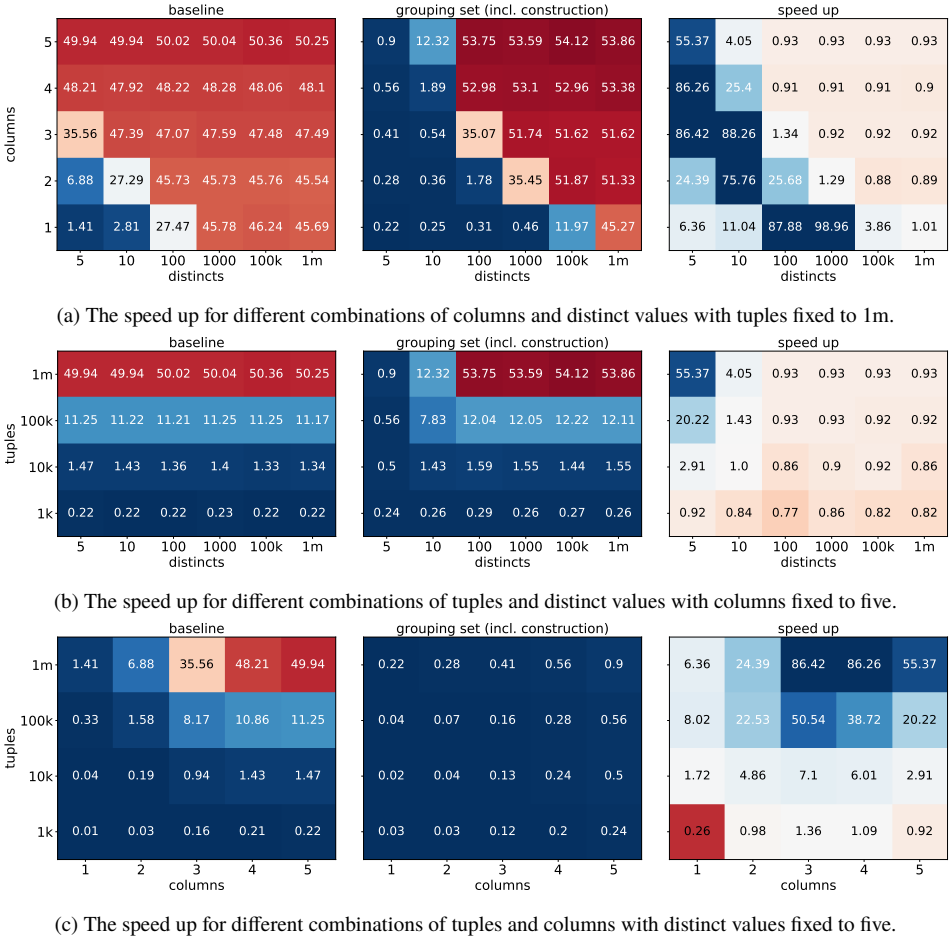


Fig. 7: Execution times and speed ups on synthetic data.

of distinct values per column equals the number of tuples in the table. If such a grouping set is instantiated, each of its dimensions has a length equal to the number of tuples. This grouping set is larger than the original data because it includes the aggregate. With all that in mind, we need to quantify the benefit of a grouping set for our use cases.

3.2 Benefit Criterion

To find a useful criterion when to instantiate grouping sets, we evaluate the usefulness of these sets on synthetic data. Again, we need to find a way to express the compression of information in a grouping set. From the synthetic data, we will derive a general rule for

the theoretical improvement of a grouping set on a given table or join. Our experiment comprises six steps per iteration. The first step generates a synthetic table given three properties. These properties are: the number of tuples in a single table or in the largest table of a join N , the number of columns C , and the number of distinct values per column D . We vary the properties in the ranges:

$$N \in \{1\,000, 10\,000, 100\,000, 1\,000\,000\} \quad (1)$$

$$C \in \{1, 2, 3, 4, 5\} \quad (2)$$

$$D \in \{5, 10, 100, 1\,000, 100\,000, 1\,000\,000\} \quad (3)$$

Changing one property per iteration, this leads to $|N| \cdot |C| \cdot |D| = 4 \cdot 5 \cdot 6 = 120$ different tables or iterations. The values in a column are uniformly sampled from the range of distinct values. With an increasing number of distinct values per column, we simulate floating point columns which have a large number of different values. Columns with few distinct values resemble categorical data. In the second step, we sample 1,000 count aggregate queries as an example workload over all possible combinations of columns (predicates), operators, and values in this iteration. In the third step, we execute these queries against the table and measure their execution time. This is equivalent to the standard procedure to sample example cardinality queries for an ML model. The fourth step constructs the grouping sets over the whole synthetic table and measures its construction time. Next, in step five, we rewrite the queries in a way that they can be executed against the grouping set. We measure their execution time on this grouping set. In the final step, we divide the execution time of the workload on the grouping set by the run time of the workload on the table. This *speed up factor* ranges from close to zero for a negative speed up to infinity for a positive speed up. All time measurements are done three times and averaged. We use PostgreSQL 10 for the necessary data management.

Figure 7 shows the results of all 120 iterations. Blue values mean either better execution times or higher speed up, whereas red means longer execution times or lower speed up. White indicates a speed up factor of one. The first column shows the execution time of the workload against the table. The next column shows the execution time of the workload against the grouping set including the construction time of the grouping set. The last column is the quotient of the second and first column. This is the achieved speed up by using a grouping set. In each row, only two properties are changed while the third property is kept fixed. The first row keeps the number of tuples, the second row the number of columns, and the last row the number of distinct values fixed. From this figure, we can derive three conclusions.

First, we notice that few distinct values in a few columns are beneficial for the aggregation. Next, the more tuples N are in a table, the more distinct values per column can be there for the speed up to be sustained. As the last conclusion, this also applies to the number of columns. All in all, the larger the original table the more distinct values and columns still lead to a speed up. Our experiments show that a grouping set is only beneficial if its size is smaller than the original table. Only then the aggregate compresses information and causes

Input : workload wl , database
Output : grouping sets

```

1 grouping sets: tables  $\rightarrow$  attributes
2 for query  $q$  in  $wl$  do                                     // Analyze
3   | tables, attributes of  $q$                                    // step 1
4   | grouping sets[tables] =
5   |   grouping sets[tables]  $\cup$  attributes
6 end

7 for grouping set  $gs$  in grouping sets do
8   |  $N = \max(|tuples| \text{ of all tables of } gs)$                 // step 2
9   | for attribute  $p$  in attributes of  $gs$  do
10  |   |  $dv = dv \cup |distinct\ values\ of\ p|$ 
11  | end
12  | scaling factor =  $\frac{\prod dv}{N}$                                 // step 3
13  | if scaling factor  $\geq 1$  then
14  |   | grouping sets[tables] = attributes where  $\frac{\prod dv}{N} < 1$ 
15  |   | construct grouping set
16 end

```

Algorithm 1: Analyze component.

less data to be scanned by the queries. Given our evaluation, this happens if the product of the distinct values of all columns is smaller than the table size. We can model this as an equation to be used as a criterion for instantiating beneficial grouping sets.

$$\text{scaling factor} = \frac{1}{N} \prod_{c=1}^C |\text{distinct_values}(\text{column}_c)| \quad (4)$$

If this *scaling factor* is smaller than one, we call a grouping set beneficial. The scaling factor is also a measure of data compression. Therefore, it shows how much faster the scan over the aggregated data can be.

4 Implementation

In this section, we describe the implementation of our *aggregate-based training phase* for ML-based cardinality estimation in DBMS in detail. In our implementation, we assume that a regular DBMS with an SQL interface provides the base data and the ML models are trained outside the DBMS, e.g., in Python. Based on this setting, we added a new layer implemented in Python² between these systems to realize our *aggregate-based training phase* in a very flexible way. Thus, the input of this layer is an ML workload that is necessary for training the ML model. Then, the main tasks of this layer are:

² We plan to make the code base publicly available in case of paper acceptance.

Input : workload wl , grouping sets

Output : rewritten workload wl'

```

1 for query  $q$  in  $wl$  do                                     // Rewrite
2   | tables, attributes of  $q$ 
3   | if  $attributes = grouping\ sets[tables]$  then
4   |   | rewrite  $q$  to match grouping set
5   |   add  $q$  to  $wl'$ 
6 end

```

Algorithm 2: Rewrite component.

1. discover as well as create as many beneficial grouping sets in the DBMS as possible for the given ML workload
2. rewrite as well as execute the workload queries according to the grouping sets and base data.

The output of this layer is an annotated ML workload with the retrieved cardinalities on which the ML model is trained afterward. To achieve that, our layer consists of two components. The first component is the Analyzer which is responsible for the construction of beneficial grouping sets. The second component is the Rewrite rewriting and executing the queries of the ML workload against the constructed grouping sets. In the following, we introduce both components in more detail.

4.1 Analyzer Component

Algorithm 1 gives a more detailed overview over the Analyzer Component. Given an ML workload and a database instance, our Analyzer consists of three steps to find and build all beneficial grouping sets. In **step one**, the Analyzer scans all queries in the ML workload and collects all joins or tables and their respective predicates in use. This generates all possible grouping sets as a mapping from tables building the grouping set to the predicates on those tables. In Algorithm 1, this is covered in lines 1 to 6. The **second step** then collects the number of distinct values per predicate attribute (regardless of their type) and the maximum number of tuples of all tables in the grouping set from the metadata (statistics) of the database. This is shown in lines 8 to 11. In the **third and final step**, our defined benefit criterion (Equation (4)) is used to calculate the scaling factor and therefore the benefit of each grouping set. If the scaling factor is smaller than one, the Analyzer constructs the grouping set with all collected predicates. If the scaling factor is larger than or equal to one, the grouping set is constructed with the maximum number of predicates where the scaling factor still is smaller than one. This may disregard certain queries that are subsequently not executed against the grouping set if they have more predicates than the grouping set. On the other hand, queries on the table or join with the predicates in the grouping set can still benefit from it. Moreover, all queries to be executed against a grouping set are marked for rewriting. This final step is detailed in lines 12 to 15.

4.2 Rewrite Component

With all beneficial grouping sets instantiated by the `Analyzer` Component, it is necessary to modify the ML workload queries to be able to use the pre-aggregates. For this, all queries which can be run against any grouping set will be rewritten in the `Rewrite` component. The `Rewrite` component receives information about each query from the `Analyzer` and rewrites queries in a way that they can be executed against the grouping sets. All queries where the `Analyzer` does not recognize a grouping set are kept as they are and will be executed over the base data. The `Rewrite` component is described in Algorithm 2.

When all queries have been processed, the optimized workload is executed as a whole on the database. If a query has been rewritten, it will be executed against the grouping set, otherwise, it will be executed against the original data. Finally, the retrieved results (i.e. cardinalities) are forwarded to the ML system to train the ML model.

5 Evaluation

To show the benefit of our novel *aggregate-based training phase*, we conducted an exhaustive experimental study with both presented types of ML models for cardinality estimation (cf. Section 2). Thus, we start this section by explaining the experimental settings followed by a description of selective results for the local as well as global ML model approaches. Afterward, we summarize the main experimental findings.

5.1 Experimental Setting

For our experiments, we used the original workloads for the local and global ML model approaches [Ki19a, Wo19a] on the IMDB data set [IM17]. The IMDB contains a snowflake database schema with several millions of tuples in both the fact and the 20 dimension tables. As already presented in Section 2.3, the global model workload contains 100,000 queries. For the local models, we used three workloads where each workload has 5,000 queries more

model	Base Data w/o Index	Base Data w/ Index	Construction GS	Execution GS	Total GS	Coverage GS
local 1	2h 30m	1h 44m	6.17s	191.34s	197.51s	100%
local 2	7h 53m	5h 10m	23.70s	205.91s	229.61s	100%
local 3	10h 12m	6h 56m	29.10s	430.36s	459.46s	100%
global full	–	4d 14h	2h 22m	20d 20h	20d 22h	100%
global opt	–	4d 14h	34m 29s	2d 11h	2d 12h	55%

Tab. 1: Execution times ML workloads (GS: grouping sets).

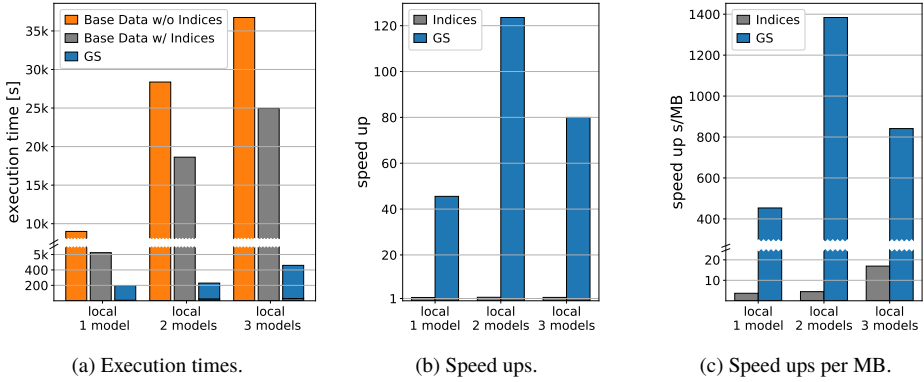


Fig. 8: Local model evaluation based on our *aggregate-based training phase* (GS: Grouping Sets).

than the previous one. These workloads correspond to one, two, and three trained local models. Overall, we have four workloads for our experiments: one for a global model and three workloads for an increasing number of local models. Moreover, all experiments are conducted on an AMD A10-7870K system with 32GB main-memory with PostgreSQL 10 as the underlying database system.

In our experiments, we measured the workload execution times, whereby we distinguish three different execution modes:

Base Data w/o Indexes: ML workload is executed on the IMDB base data without any indexes on the base data.

Base Data w/ Indexes: ML workload is executed on the IMDB base data with indexes on all (join) predicates in use.

Grouping Set (GS): ML workload is executed on *pre-aggregated data* as determined by our approach.

The first two execution modes represent our baselines because both are currently used in the presented ML model approaches for cardinality estimation [Ki19b, Wo19b, Li15].

5.2 Experimental Results: Local Model Workloads

Figure 8 shows the results for the three local model workloads. The first workload contains the necessary queries to build one local ML model to estimate the cardinalities for the join `title>movie_keyword`. The second workload adds 5,000 queries to the first workload to construct a second ML model for the join `title>movie_info`. The third workload adds another ML model for an additional join `title>movie_companies`. Therefore, we increment the number of local ML models.

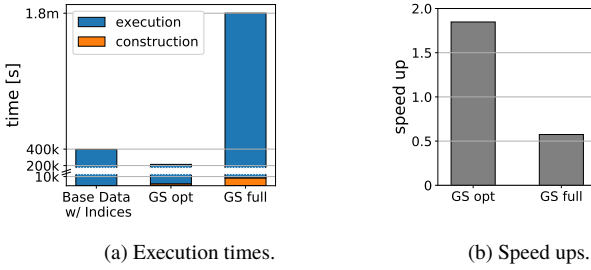


Fig. 9: Global model evaluation results.

Figure 8a details the execution times for all three local model workloads for all investigated execution modes. In each of the three groups, the left bar shows the complete workload execution time on the IMDB without indexes, the middle bar on the IMDB with indexes, and the right bar the execution time with our grouping set approach. As we can see, indexes on the base data are already helping to reduce the workload execution times compared to execution on the base data w/o indexes, but the speedup is very marginal as shown in Figure 8b because the DBMS might decide to abstain from using the indexes. In contrast to that, our grouping set approach has the lowest execution times in all cases and the achieved speed ups compared to execution on the base data w/o indexes are in the range between 45 and 125 as depicted in Figure 8b. Thus, we can conclude that our *aggregation-based training phase* is much more efficient than state-of-the-art approaches.

For each considered join, our *aggregation-based* approach creates a specific grouping set containing all columns from the corresponding workload queries. According to equation (4), the scaling factors are: 0.02, 0.003, and 0.05 for the three joins. Thus, the instantiation of grouping sets is beneficial. So, all grouping sets achieve a very good compression rate and the rewritten workload queries on the grouping sets have to read much less data compared to the execution on base data. Moreover, all workload queries can be rewritten, so that the coverage is 100% and every query benefits from this optimization. Nevertheless, the three scaling factors differ explaining the different speed ups.

The construction of the grouping sets can be considered a drawback. However, as illustrated in Table 1, the construction times for the grouping sets are negligible because the reduction in execution time is significantly higher. From a storage perspective, index structures and grouping sets need some extra storage space, where the storage overhead for grouping sets is larger than for indices. But, as illustrated in Figure 8c, the speed up per additional MB for grouping sets is much larger than for indices. All in all, we gain a much larger speed up making grouping sets the more efficient approach.

5.3 Experimental Results: Global Model Workload

Figure 9 shows the evaluation results in terms of execution and construction times for the global model workload. As shown in the previous experiment, the utilization of indices is always beneficial. Thus, we only compare the execution on base data with indices and the execution on the aggregated data in this evaluation.

In general, there are 21 grouping sets possible for the global workload. However, some of these grouping sets have a scaling factor larger than one. Therefore, our Analyzer component disregards the attributes of some grouping sets until the scaling factor is smaller than one (cf. Section 4). As a consequence, only 55% of the global workload queries can be rewritten to this optimal set of grouping sets. This strategy called *GS opt* in Figure 9a reduces the workload execution time of the global workload. The speed up compared to the execution on the base data with indexes is almost 2 (Figure 9b).

To show the benefit of our grouping set selection strategy, we also constructed all grouping sets with all attributes (*GS full*). There, we are able to rewrite all global workload queries to be executed on these aggregated data. As shown in Figure 9a, the overall workload execution time is longer than the execution on base data with indices. Therefore, grouping sets have to be selected carefully. Moreover, this experiment shows that our definition of a beneficial grouping set is applicable because (i) not all grouping sets are beneficial and (ii) not all queries can or need to be optimized with a grouping set. The benefit criterion considers both aspects to reduce workload execution times.

5.4 Main Findings

For both types of ML models for cardinality estimation, our *aggregation-based training phase* offers a database-centric way to reduce execution time. Table 1 summarizes our evaluation results. The overhead introduced by the construction of a grouping set is much smaller than the savings in execution time. So, grouping sets reduce the workload execution times and amortize their own construction time. The simpler structure of the local model workloads is better supported by grouping sets because they contain fewer combinations of columns and fewer distinct values. These are exactly two of the assets for grouping sets identified in Section 3. This leads to a higher performance speed up for local model workloads than for global model workloads with consistent high-quality estimates. Thus, we can afford a larger amount of local models to reach the schema coverage of a global model. Even if these models request more queries than the global model, their benefits from the use of grouping sets outweigh the higher number of queries.

6 Related Work

In this section, we detail the importance of database support for machine learning in other works. We look at the motivation for pre-aggregates from both the database system and the machine learning point of view.

When looking at the synergy of database and machine learning systems, there are three possible interactions: (i) integrate machine learning into database systems, (ii) adapt database techniques for machine learning models, and (iii) combine database and machine learning into one life cycle system [KBY17]. Based on that, we classify our work in category (iii). However, the focus in this area is more on feature and model selection and not on sampling example data. We argue that the direct support of machine learning training phases with databases should be treated with the same attention.

To the best of our knowledge, there is only little research on directly optimizing the sampling of workloads for machine learning problems. The authors of [Ki19b] detail their method of speeding up query sampling in [Ki19c]. They use massive parallelism by distributing the workload over several DB instances. We see this as a promising step because our approach can also profit from parallel execution. Especially the instantiation and the querying of grouping sets can be done in parallel because grouping sets are orthogonal to each other.

Another thing to look at is the availability of supportive data structures in database systems. The cube operators are established in databases and benefit from a wide-ranged support [Ag96, Gr96, HRU96, Sh96, ZDN97]. The ability of a database to deliver necessary meta information is also important. For example, fast querying for the distinct values of each column has a large impact on performance. A simple solution for this is a dictionary encoding of the data in the database. Some database systems already use dictionary coding for all their data [Fä12]. This is beneficial for our approach because from a dictionary encoded column it is easy to yield the number of distinct values with a dictionary scan. Moreover, dictionary encoding directly supports the transformation of the data into the grouping set dimension and the definition of ranges of these dimensions.

Aside from machine learning, database support for data mining has already been an important research topic [AS96, CWC09, HLH03, Ne01, OC00]. For example, [HLH03] identifies that aggregation in sub-spaces formed by combinations of attributes is a common task in many data mining algorithms. Based on that observation, we see a large potential for tighter coupling of databases and mining algorithms.

7 Conclusion

We made the case for cardinality estimation as a candidate for database support of machine learning for DBMS. We detailed an approach for pre-aggregating count information for cardinality estimation workloads. It uses grouping sets, a well-known database operator, to

reduce the data to be scanned by example queries for cardinality estimation with machine learning models. This reduces the execution time of a given workload even though we spend extra time to construct the intermediate data structures.

This case has a strong potential to be applied to the other similar machine learning problems, like plan cost modeling or indexing. We see parallels between the potential for machine learning workloads and any of these machine learning problems where information about the data in the DB is aggregated. These parallels make grouping sets and therefore DB support beneficial for ML for DBMS in general.

Bibliography

- [Ag96] Agarwal, Sameet; Agrawal, Rakesh; Deshpande, Prasad; Gupta, Ashish; Naughton, Jeffrey F.; Ramakrishnan, Raghu; Sarawagi, Sunita: On the Computation of Multidimensional Aggregates. In: VLDB. pp. 506–521, 1996.
- [AS96] Agrawal, Rakesh; Shim, Kyuseok: Developing Tightly-Coupled Data Mining Applications on a Relational Database System. In: KDD. pp. 287–290, 1996.
- [CWC09] Cho, Chung-Wen; Wu, Yi-Hung; Chen, Arbee L. P.: Effective database transformation and efficient support computation for mining sequential patterns. *J. Intell. Inf. Syst.*, 32(1):23–51, 2009.
- [Da19] Damme, Patrick; Ungethüm, Annett; Hildebrandt, Juliana; Habich, Dirk; Lehner, Wolfgang: From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM TODS*, 44(3):9:1–9:46, 2019.
- [Fä12] Färber, Franz; May, Norman; Lehner, Wolfgang; Große, Philipp; Müller, Ingo; Rauhe, Hannes; Dees, Jonathan: The SAP HANA database - An architecture overview. *IEEE Data Eng. Bull.*, 35:28–33, 03 2012.
- [FM11] Fender, Pit; Moerkotte, Guido: A new, highly efficient, and easy to implement top-down join enumeration algorithm. In: ICDE. pp. 864–875, 2011.
- [Gr96] Gray, J.; Bosworth, A.; Lyaman, A.; Pirahesh, H.: Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In: ICDE. 1996.
- [HLH03] Hinneburg, Alexander; Lehner, Wolfgang; Habich, Dirk: COMBI-Operator: Database Support for Data Mining Applications. In: VLDB. pp. 429–439, 2003.
- [HN17] Harmouch, Hazar; Naumann, Felix: Cardinality Estimation: An Experimental Survey. *PVLDB*, 11(4):499–512, 2017.
- [HRU96] Harinarayan, Venky; Rajaraman, Anand; Ullman, Jeffrey D.: Implementing Data Cubes Efficiently. In: SIGMOD. pp. 205–216, 1996.
- [IM17] IMDB: Internet Movie Database. <ftp://ftp.fu-berlin.de/pub/misc/movies/database/frozendata/>, Accessed on 2020-06-01.
- [KBY17] Kumar, Arun; Boehm, Matthias; Yang, Jun: Data management in machine learning: Challenges, techniques, and systems. In: SIGMOD. pp. 1717–1722, 2017.

- [KHL17] Karnagel, Tomas; Habich, Dirk; Lehner, Wolfgang: Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *PVLDB*, 10(7):733–744, 2017.
- [Ki19a] Kipf, Andreas: Learned Cardinalities in PyTorch. <https://github.com/andreaskipf/learnedcardinalities/>, Accessed on 2020-08-20.
- [Ki19b] Kipf, Andreas; Kipf, Thomas; Radke, Bernhard; Leis, Viktor; Boncz, Peter A.; Kemper, Alfons: Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In: *CIDR*. 2019.
- [Ki19c] Kipf, Andreas; Vorona, Dimitri; Müller, Jonas; Kipf, Thomas; Radke, Bernhard; Leis, Viktor; Boncz, Peter; Neumann, Thomas; Kemper, Alfons: Estimating Cardinalities with Deep Sketches. In: *SIGMOD*. p. 1937–1940, 2019.
- [Kr18] Kraska, Tim; Beutel, Alex; Chi, Ed H; Dean, Jeffrey; Polyzotis, Neoklis: The case for learned index structures. In: *SIGMOD*. pp. 489–504, 2018.
- [Le15] Leis, Viktor; Gubichev, Andrey; Mirchev, Atanas; Boncz, Peter; Kemper, Alfons; Neumann, Thomas: How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.
- [Li15] Liu, Henry; Xu, Mingbin; Yu, Ziting; Corvinelli, Vincent; Zuzarte, Calisto: Cardinality Estimation Using Neural Networks. In: *CASCON*. pp. 53–59, 2015.
- [MNS09] Moerkotte, Guido; Neumann, Thomas; Steidl, Gabriele: Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB*, 2(1):982–993, 2009.
- [MP19] Marcus, Ryan; Papaemmanouil, Olga: Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB*, 12(11):1733–1746, 2019.
- [Ne01] Netz, Amir; Chaudhuri, Surajit; Fayyad, Usama M.; Bernhardt, Jeff: Integrating Data Mining with SQL Databases: OLE DB for Data Mining. In: *ICDE*. pp. 379–387, 2001.
- [OC00] Ordóñez, Carlos; Cereghini, Paul: SQLEM: Fast Clustering in SQL using the EM Algorithm. In: *SIGMOD*. pp. 559–570, 2000.
- [Ro15] Rosenfeld, Viktor; Heimerl, Max; Viebig, Christoph; Markl, Volker: The Operator Variant Selection Problem on Heterogeneous Hardware. In: *ADMS@VLDB*. pp. 1–12, 2015.
- [Sh96] Shukla, Amit; Deshpande, Prasad; Naughton, Jeffrey F; Ramasamy, Karthikeyan: Storage estimation for multidimensional aggregates in the presence of hierarchies. In: *VLDB*. pp. 522–531, 1996.
- [SL19] Sun, Ji; Li, Guoliang: An End-to-End Learning-Based Cost Estimator. *PVLDB*, 13(3):307–319, 2019.
- [Wo19a] Woltmann, Lucas: Cardinality Estimation with Local Deep Learning Models. <https://github.com/lucaswo/cardest/>, Accessed on 2020-08-20.
- [Wo19b] Woltmann, Lucas; Hartmann, Claudio; Thiele, Maik; Habich, Dirk; Lehner, Wolfgang: Cardinality Estimation with Local Deep Learning Models. In: *aiDM*. ACM, 2019.
- [YW79] Youssefi, Karel; Wong, Eugene: Query Processing in a Relational Database Management System. In: *VLDB*. pp. 409–417, 1979.
- [ZDN97] Zhao, Yihong; Deshpande, Prasad; Naughton, Jeffrey F.: An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In: *SIGMOD*. pp. 159–170, 1997.

Using FALCES against bias in automated decisions by integrating fairness in dynamic model ensembles

Nico Lässig¹, Sarah Oppold², Melanie Herschel³



Abstract: As regularly reported in the media, automated classifications and decisions based on machine learning models can cause unfair treatment of certain groups of a general population. Classically, the machine learning models are designed to make highly accurate decisions in general. When one machine learning model is not sufficient to define the possibly complex boundary between classes, multiple “specialized” models are used within a model ensemble to further boost accuracy. In particular, *dynamic model ensembles* pick the most accurate model for each query object, by applying the model that performed best on similar data. Given the labeled data on which models are trained, it is not surprising that any *bias* possibly present in the data will reflect in the classifiers using the models. To mitigate this, recent work has proposed *fair model ensembles*, that instead of focusing (solely) on accuracy also optimize *global fairness*, which is quantified using bias metrics. However, such global fairness that globally minimizes bias may exhibit imbalances in different regions of the data, e.g., caused by some local bias maxima leading to *local unfairness*. In this paper, we propose to bridge the gap between dynamic model ensembles and fair model ensembles and investigate the problem of devising locally fair and accurate dynamic model ensembles, which ultimately optimize for equal opportunity of similar subjects. Our evaluation shows that our approach outperforms the state-of-the-art for different types and degrees of bias present in training data in terms of both local and global fairness, while reaching comparable accuracy.

Keywords: Model fairness; bias in machine learning; model ensembles

1 Introduction

In decision support systems (DSS), machine learning models are frequently used to make recommendations or even decisions. While these unquestionably simplify many processes and tasks arising in modern life, critical situations arise in automatic classification scenarios such as credit scoring, or predictive policing applications. There, critical DSS automatically assign people to classes that have the possibility to deeply affect their lives in a positive or negative way. Recent real-life examples where the use of such DSS had to be revoked due to underlying biased classifiers include an algorithm that determined A-Level grades of British students who were unable to take their exams due to COVID-19 regulations [Co20]. Based on the teachers’ assessment of the student’s performance and the school’s performance in subjects, each student was assigned A-Level grades. Using these features, about 40% of

¹ Universität Stuttgart, nico.laessig@ipvs.uni-stuttgart.de

² Universität Stuttgart, sarah.oppold@ipvs.uni-stuttgart.de

³ Universität Stuttgart and University of Singapore, melanie.herschel@ipvs.uni-stuttgart.de

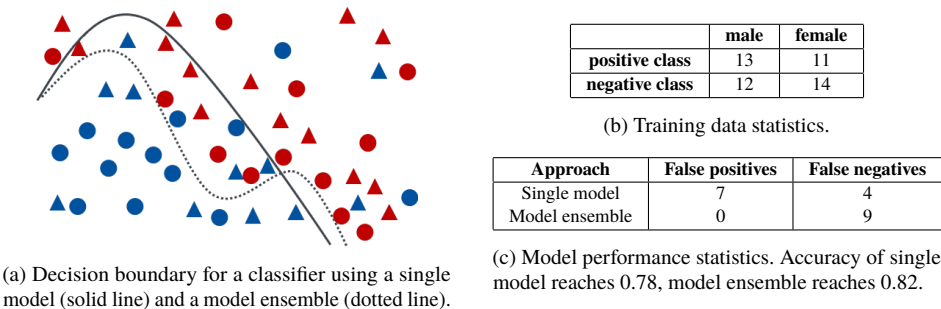
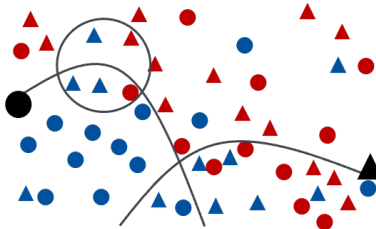


Fig. 1: Example binary classification scenario deciding about employee raises.

British students received lower grades than recommended by their teachers, as the model indirectly favored students from private schools and wealthy areas. After a public outcry, the algorithmic decisions were revoked and replaced by the teachers’ assessments. Another example is a recruitment tool developed by Amazon [Da18]. The tool was supposed to automatically assign scores to job applicants based on their application to support making hiring decisions. However, it exhibited discrimination against women, a problem that could not be resolved, leading to the project being discarded after several years of investment.

Classification tasks performed by DSS are, by themselves, not trivial to solve. For instance, consider Figure 1, which summarizes a simple classification problem when deciding on employee salary raises. We visualize the training data in Figure 1a, where we place similar employees close to each other and use different shapes to distinguish male (circle) and female (triangle) employees. The goal of the trained classifier is to divide in two classes, which we distinguish by color: employees in blue have a positive outcome and get a raise, while employees labeled in red are associated to the negative class (no raise). Opting for a simple classifier, let us assume we obtain the decision border shown as solid black line in Figure 1a. It classifies all employees below the line into category “blue” and all persons above the line into category “red”. Using this simple classifier, a number of people are assigned to the wrong class (see Figure 1c). We see that the simple classifier yields an accuracy of 0.78, computed as the fraction of correctly classified points vs all data points. To obtain a classifier that more faithfully reflects the complex decision boundary in our example and thus improves accuracy, we can resort to *model ensembles*. There, different (simple) models are trained and then combined, e.g., to reach a classifier with the decision border shown as a dashed line in Figure 1a. This allows us to improve the accuracy from 0.78 to 0.82 in our example.

While the above example illustrates one means to boost the accuracy of classifiers, it leaves aside any consideration of *fairness*. The term fairness is often used in the literature to refer to non-discrimination. In the introductory examples, we see that not all students or job applicants were treated equally and some discrimination was unintentionally introduced to the classifiers. Such unfair behavior is commonly linked to some *bias*. There are many



(a) Decision boundary for a fair model ensemble that combines classifiers for male (left) and female employees (right) and illustration of a locally unfair situation (circle).

Approach	Accuracy	Fairness
Single model	0.78	0.76
Model ensemble	0.82	0.66
Fair model ensemble	0.78	0.81

(b) Model performance statistics.

Fig. 2: Example binary classification scenario deciding about employee raises (Figure 1 continued).

different kinds of bias that can be introduced through the data or human decisions. For instance, while it may seem reasonable to consider student’s past performance as a feature, e.g., on mock-exams to assign a final grade, wealthy students who benefit from regular tutoring may be at an advantage over students that learn for exams on their own. In case of automatic resume analysis, having a training dataset with CVs predominantly from male applicants possibly causes models that favor terms more commonly used by men than women while penalizing terms associated to women. Returning to our fictional example, based on the numbers reported in Figure 1b, we see that the training data is reasonably balanced in terms of men and women being assigned a positive or negative label, which is a good starting point. To assess the classifiers in terms of fairness, we can use one of many available bias metrics. The American Title VII of the Civil Rights Act of 1964 prohibits employment discrimination and, for example, states that there is discrimination when the probability of a woman getting a positive outcome divided by the probability of a man getting a positive outcome is less than 0.8. In the case of the single classifier and model ensemble, the value is 0.76 and 0.66 respectively (see Figure 2b) and thus below the threshold. So using these classifiers would be against the law in the US.

With metrics quantifying bias being available, recent approaches have considered these to prevent bias. In particular, Dwork et al. [Dw18] have introduced *fair model ensembles*. Given a pre-defined set of sensitive groups (e.g., women), their approach learns classifiers dedicated to these groups and then calculates the best combination of classifiers according to a metric that combines both fairness and accuracy. By training classifiers specialized to different groups, the approach can better capture different patterns exhibited by different groups. Optimizing for both fairness and accuracy compromises between fair treatment of the different groups and model performance. Figure 2 illustrates the effect of using the method for fair model ensembles in our example. It combines two classifiers, for which we show the decision borders: one trained for male employees (left hand side) and one for women (right hand side). Instead of a fairness of 0.66, the positive classification of negatively labeled women by the dedicated classifier raises fairness to 0.81 and therefore (at least according to American law) no longer exhibits discrimination.

While the approach illustrated above comes closer to the goal of treating members of different predefined groups (e.g., male, female) equally, it does so from a global perspective. Thus, localized inequalities remain. For instance, looking again at Figure 2a, while the goal of non-discrimination between women and men is met, the subregion within the depicted circle exhibits local unfairness. As a reminder, we have placed persons in the 2-dimensional representation close to each other when they have similar features, e.g., all persons in the circled region may have similar age and number of sick days. Clearly, despite similar features, women in this region are significantly less likely to get a raise than men. This corresponds to a *local bias*. The approach presented in this paper aims at solving this issue.

The fact that optimization goals are only fulfilled globally and not locally is not a peculiarity of fairness. *Dynamic model ensembles* tackle this problem when optimizing accuracy. Intuitively, a model or model ensemble is dynamically selected for each new decision based on model performance in similar situations. This paper uses a similar rationale to optimize the overall local fairness of a model ensemble by combining ideas of both fair model ensembles and dynamic model ensembles. More precisely, our contributions are as follows:

- We present a framework for addressing the novel problem of mitigating locally unfair decisions. In an offline phase, it trains a diverse set of models to get accurate and fair models for different groups. In an online phase, it dynamically selects the model ensemble best suited for the different groups when focusing on group members similar to the subject to be classified. This combines ideas previously devised for (static) fair model ensembles and dynamic model ensembles specialized on accuracy.
- We present FALCES, which implements our framework using several algorithms. It comes in different variants, depending on whether the training data is further split before training classifiers or if trained classifiers are further pruned based on an initial assessment of their global combined performance in terms of fairness and accuracy. It also relies on a novel metric for a combined quantification of fairness and accuracy.
- We implement our algorithm variants and evaluate them on both synthetic and real data. Our results show that while we cannot fully eliminate bias, FALCES outperforms the state-of-the-art in both global group fairness and local group fairness, the latter quantified using a newly defined metric for local fairness. At the same time, our solution does not jeopardize accuracy.

The remainder of this paper is structured as follows. Section 2 reviews related work. We present our framework in Section 3 and discuss algorithms implementing the framework in Section 4. Our implementation and experimental evaluation are presented in Section 5. The paper concludes with a summary and outlook on future research in Section 6.

2 Related work

Our proposed solution builds on previous work on model ensembles and fairness in machine learning, in particular fair model ensembles and dynamic model ensembles.

2.1 Model ensembles

The idea of model ensembles is to train multiple models and select or combine the best of these models [Po06]. Hereby, the optimization goal typically is the improvement of the accuracy of predictions [SHX19, DSM19, Zh19]. Combining the outputs of several models has proven to be preferable compared to single-model systems. By combining the results of several models, model ensembles can, for example, reduce the risk of choosing a model that performs poorly, which reduces the overall risk of a bad decision, or overcome complex decision borders that may not be able to be implemented by a chosen model because they lie outside the functional space of the model. The same rationale underlies fair model ensembles (discussed further below), which set an additional optimization focus on increasing fairness.

2.2 Fairness in machine learning

As already described in the introduction, the term fairness in machine learning commonly refers to the fact that models must not discriminate against people because of bias(es). Based on various laws, social definitions and understood meanings, different measures to quantify fairness have been defined [KC09, PRT08, Ž117]. They can be broadly classified into two categories. A group of metrics for *individual fairness* (or equality or equality of treatment) focuses on providing equal treatment to equal people [FSV16]. However, we will focus on the second notion of fairness: *group fairness*. It is also known as equality of outcome or equity. Here, groups with different preconditions are treated differently, so that in the end everyone, despite their differences, has the same opportunities. This is intended to overcome social inequalities and offer equal opportunity to different groups [FSV16].

Based on these fairness metrics, methods have been developed which allow the development of individual fair models using fair data, new machine learning algorithms, or techniques for modifying existing models [KC09, PRT08, Ga19].

2.3 Fair model ensembles

While there is now a visible body of research on measuring fairness and learning single fair models, only few works leverage multiple models in order to achieve fairness, thereby bringing the advantages of using model ensembles to the the realm of improving fairness.

Calders and Verwer [CV10] create fair naive Bayes model ensembles. To this end, they split the dataset according to the favored and discriminated groups and learn a naive Bayes model on each subset with the intention to classify independently of a given sensitive attribute. An overall naive Bayes model chooses the decision of either model depending on the group of incoming data tuples to be classified. While this approach yields fairer models, it is specialized to and does not extend beyond naive Bayes models.

Dwork et al. [Dw18] combine multiple machine learning classifiers to improve group fairness. They provide different versions of their algorithm, where the models are either learned on the different subgroups as in [CV10] or on larger data subsets in order to prevent accuracy loss. Their approach uses a joint loss metric that optimizes for both accuracy and fairness in order to assess which model should be used for which group of the dataset. While this approach does consider both accuracy and fairness at group level using any type of classifier, it may suffer from local unfairness.

2.4 Dynamic model ensembles

Dynamic classifier selection [CSC18] selects one classifier during runtime for each new sample which has to be classified. This is based on the rationale of model ensembles that not every classifier is an expert in all local regions of the feature space. Usually, for each new sample the local region is first identified, for example using k-nearest-neighbors (kNN). Then, the quality of available classifiers is determined and the best one(s) are selected. Dynamic ensemble selection is similar, it merely selects ensembles instead of classifiers. One example is the Dynamic Classifier Selection by Local Accuracy (DCS-LA) algorithm by Woods et al. [WKB97]. First, it uses kNN to identify the local region. Then, local accuracy of classifiers is evaluated as percentage of correctly classified samples in the local region. Alternatively, it uses local class accuracy, which is the accuracy of classifiers in the local regions with respect to the class the classifiers assign to the new sample. Only the most accurate classifier is then used to classify the unknown sample.

3 Framework for fair and dynamic model ensembles

As motivated in the introduction, our goal is to combine the benefits of fair model ensembles on the one hand and dynamic model ensembles on the other hand to devise a solution that resolves not only global bias among different groups, but also local bias, while not compromising overall accuracy. The rationale is that, while it is typically possible to define groups that should be treated fairly (and that are often defined by law), it is quite challenging to fully anticipate variations (sub-groups) within these groups that indirectly cause local bias. Techniques to counter local bias can potentially help in reaching equal opportunity among groups with similar features or similar trajectory. In this section, we first formalize our problem statement to counter locally unfair decisions. We then present a framework where we combine the ideas of fair and dynamic model ensembles to solve this problem.

3.1 Locally unfair decisions

As illustrated in Figure 2a, the problem with locally unjust decisions is that while existing solutions (reviewed in Section 2) are optimized to make globally fair and accurate decisions, there are still local regions where data points of different groups are treated unfairly. To address this problem, the decision should be optimized so that the optimal (i.e. fair and accurate) decision can be made at a local level. Our emphasis in this paper is on *group fairness*, i.e., equal opportunities between groups.

Formally, we consider as given a labeled data set D , a similarity metric s , a positive integer k , an optimization metric af combining fairness and accuracy, and a set of machine learning models (classifiers) M for the same classification task. Furthermore, D can be partitioned into groups G , for which equal opportunity is relevant. We further assume a new test sample t that belongs to one of the groups G . Then, we define our goal of *locally fair and accurate classification* as a classification task that classifies t using a model $m \in M$ with the best performance according to the af metric in the local region of D around t . This local region includes the k items in D most similar to t according to s .

3.2 Framework

To address the problem defined above, we combine the rationales underlying both fair and dynamic model ensembles described in Section 2 into a new framework for fair and dynamic model ensembles. The main components of this framework are visualized in Figure 3. We distinguish between an *offline phase* (bottom part), where suited model ensembles are trained and selected, and an *online phase* (upper part), where a previously unseen test sample t is classified by dynamically selecting the model ensemble most appropriate for t .

Offline phase. The first step of the offline phase, named *model training*, is a step common to model ensemble approaches. Here, using training data taken from the labeled dataset D , a diverse set of classifiers is trained. Given that we target both fair and accurate decisions, model training can benefit from using different subsets of data based on the different groups G present in D , which has for instance been proposed for fair model ensembles (see Section 2). We denote the set of classifiers resulting from model training considering different groups G as $M = \{(m_1, g_1), (m_2, g_2), \dots, (m_n, g_n)\}$, where each m_i is a model and g_i identifies the group it is trained for. Among these classifiers, not all may be suited to make both fair and accurate decisions. Also, too many classifiers to be considered during the online phase (further discussed below) can be computationally prohibitive. Therefore, during *model pruning*, the framework assesses all model combinations or ensembles possible with the classifiers in M that use exactly one classifier per group g_i . Assessment is done with respect to the global performance metric af that considers both accuracy and fairness. Only the best classifier combinations are retained after model pruning, resulting in $MC = \{[(m_{11}, g_1) \dots (m_{1n}, g_n)], \dots, [(m_{en}, g_1) \dots (m_{en}, g_n)]\}$, a set of model ensembles (in square brackets) s.t. for each ensemble, $g_i \neq g_j$ when $i \neq j$ and $n = |G|$.

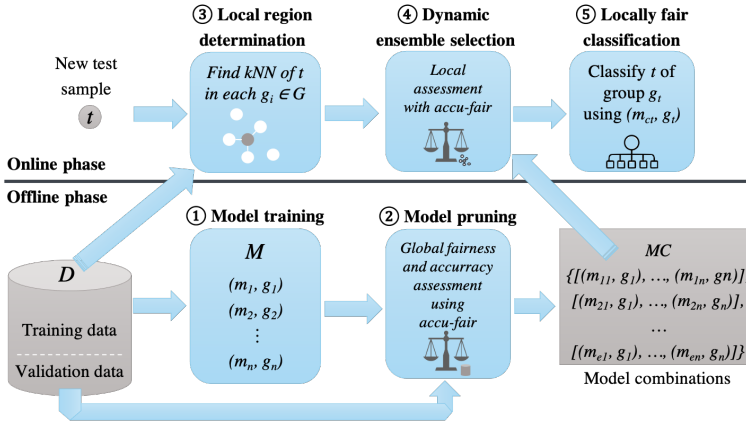


Fig. 3: Framework for locally fair classifications by combining fair and dynamic model ensembles

Online phase. When a new test sample t is to be classified, the framework determines the local region t belongs to as part of *local region determination*. To this end, it performs a kNN search of t on each g_i , where g_i is a group in $G = \{g_1, \dots, g_n\}$. The framework specifically selects an equal number of members similar to t of each group, to have a locally balanced data region with respect to the different groups. Then, for this particular region, *dynamic ensemble selection* assesses which ensemble $E \in MC$ achieves the best local performance with respect to af . Intuitively, this dynamically selects the optimal model ensemble comprising a dedicated model for each group for the region most relevant to t . With this approach, our framework combines previous techniques separately devised for fair and dynamic model ensembles. The identification of the locally best model is performed according to dynamic model ensemble techniques using fair model ensemble metrics. Therefore the classifiers are tested on the local region of the test sample using an af metric. Finally, the best classifier m_{ct} such that $(m_{ct}, g_t) \in E$ and g_t corresponds to the group t belongs to is used in the final step of *locally fair classification* to classify t .

4 Algorithms implementing the framework

Section 3 discussed the general framework that we propose for locally fair and accurate classifications. There are a variety of techniques from both fair and dynamic model ensemble research, which can be applied or extended to implement its components. In the following, we discuss the algorithms we consider to implement the framework that stand behind our FALCES system (standing for Fair and Accurate Local Classifications using EnsembleS).

4.1 Model training

As mentioned before, the set of classifiers should be diverse in order to benefit from combining them to model ensembles. To this end, we vary both the set of machine learning techniques used to train classifiers as well as the data from D that is considered for training.

In principle, any machine learning technique suited for classification tasks can be considered as candidate technique. In our evaluation, we will resort to simple techniques, e.g., decision trees, logistic regression, or nonlinear support vector machines.

Concerning the data, following previous research on fair model ensembles [Dw18, CV10], we consider splitting the input dataset D on pre-defined sub-datasets that correspond to the different groups for which we aim to achieve group fairness (e.g., we divide by sex (male, female) and race (white, others) in our experiments which creates four subgroups). This effectively partitions D into $G = \{g_1, \dots, g_n\}$, assuming n distinct groups. Then, models are trained separately for the different partitions. Different training datasets not only have the advantage of learning different models that exhibit their strengths in certain areas of the feature space. Indeed, as shown in Calders et al [CV10], the label does not depend directly on the sensitive group. In addition, complex decision borders between the two groups, which originate from different behavioral patterns, can be better modeled, thus increasing accuracy [Dw18].

As a result, similar to [Dw18, CV10], we obtain classifiers that are “specialized” on some group. More precisely, in this variant, we obtain $M = \{(m_1, g_1), \dots, (m_k, g_n)\}$ where each (m_i, g_j) associates a classifier m_i to a group g_j . For any two $(m_i, g_j), (m_{i'}, g_{j'})$, it holds that $m_i \neq m_{i'}$, and $g_j, g_{j'} \in G$, but it is possible that $g_j = g_{j'}$.

Example 1. *As a simple example, consider we split a sample dataset following the gender of employees, which results in a group for each gender, e.g., g_F for female employees and g_M for male employees. Assuming m_1, m_2, m_3 are trained on g_M and m_4, m_5, m_6 on g_F , we obtain $M = \{(m_1, g_M), (m_2, g_M), (m_3, g_M), (m_4, g_F), (m_5, g_F), (m_6, g_F)\}$.*

On the downside, splitting the data as described above can lead to a too small dataset to train on, which often results in loss of accuracy for the classifiers. Hence, we further consider the option of training models on the complete dataset D rather than on individual partitions of G . In this setting, we have $M = \{(m_1, g_D), \dots, (m_n, g_D)\}$, where $g_D = D$.

To distinguish the two variants for implementing model training described above in FALCES, we will append a suffix SBT (for Split Before Training) for the first option, while absence of this suffix indicates training is performed on the full training data set.

4.2 Model pruning

In the offline phase, the number of classifiers can already be reduced based on their global performance in order to improve the efficiency of the online phase later. Indeed, the less classifiers need to be considered in the online phase, the faster the classification of a new test item is. As we shall see in the evaluation (Section 5.4), this has only little impact on making locally fair and accurate decisions, while runtime may improve significantly.

To assess the performance of a model when considering both accuracy and fairness, we rely on a metric that combines these two dimensions, denoted as *af*. To the best of our knowledge, the state-of-the-art metric that accounts for both accuracy and fairness is the metric proposed by Dwork et al. [Dw18] for fair model ensembles, defined as follows.

$$\hat{L} = \underbrace{\frac{\lambda}{|D|} \sum_{t_i \in D} |y_i - z_i|}_{\text{Inaccuracy}} + \underbrace{\frac{1-\lambda}{|D|} \sum_{g_j \in G} \left| \sum_{\substack{t_i \in D: \\ g_{t_i} = g_j}} z_i - \frac{1}{|G|} \sum_{t_i \in D} z_i \right|}_{\text{Unfairness}} \quad (1)$$

Here, the number of tuples in a labeled dataset D is denoted as $|D|$, each tuple $t_i \in D$ has an actual and predicted label denoted as y_i and z_i respectively, $|G|$ represents the number of different groups in G , $g_{t_i} \in G$ represents the group a tuple t_i belongs to, and $0 \leq \lambda \leq 1$ balances the relative weight of the accuracy and the fairness part of the equation. Intuitively, in the first part of the metric, accuracy is measured by comparing the predicted and actual label for each data tuple (also known as L_1 loss). The second part of the metric determines the fairness of the classifier combination that associates a classifier to each group. It sums up the difference between the sum of predicted values of each group and the overall sum of labels divided by number of groups. Note that for both sides, higher values actually mean a poorer performance, we thus qualify them as inaccuracy and unfairness.

This metric combines both accuracy and fairness, however, the fairness-part is sensitive to differences in the relative size of groups. Assume for instance there is a larger group g_L and a smaller group g_S with equal sum of z_i , i.e. equal number of positively classified tuples. Indeed, the metric considers the situation to be fair among these two groups (unfairness-part drops to 0), even though the probability that a member of g_L is assigned a positive label is smaller than the probability of a member of g_S being assigned a positive label. While this may well serve minorities that are considered protected groups and are thus indirectly favored by being part of the smaller group, it does not accurately reflect equal opportunity.

We propose a variation of *af* that modifies the fairness-related part of Equation 1 to also consider the number of tuples $|g_j|$ in a group $g_j \in G$. This results in the following metric.

$$\hat{L}_{new} = \frac{\lambda}{|D|} \sum_{t_i \in D} |y_i - z_i| + \frac{1-\lambda}{|G|} \sum_{g_j \in G} \left| \sum_{\substack{t_i \in D: \\ g_{t_i} = g_j}} \frac{z_i}{|g_j|} - \frac{1}{|D|} \sum_{t_i \in D} z_i \right| \quad (2)$$

While the accuracy part still determines L_1 loss, the fairness part has slightly changed. For each group, its mean value is set in relation to its group size and compared to the overall mean value of positive predicted labels. These are then again summed up and divided by the number of groups to allow for an arbitrary number of groups.

Using the metric proposed in Equation 2, model pruning aims at retaining only a “good” selection of ensembles formed by models of M obtained during model training. Given that we are using model ensembles, this evaluation of model quality is performed by considering all possible combinations of classifiers in M when choosing one classifier per group, and keeping only the best ones. In our implementation, we keep ensembles up to a predefined rank. Another possibility would be to use a threshold for the maximally acceptable af score.

Example 2. Continuing our previous example, given that we have three classifiers dedicated to g_F and g_M , respectively, we have a total of 9 combinations to test using af . Let us assume that the top-2 ensembles according to af are $(m_1, m_4), (m_2, m_5)$. Assuming FALCES is configured to the top-2 combinations, we obtain $MC = \{(m_1, g_M), (m_4, g_F), [(m_2, g_M), (m_5, g_F)]\}$.

Similarly to model pruning, we consider running FALCES with or without model pruning enabled. When active, we append PFA to the algorithm name.

4.3 Local region determination

Moving on to the online phase, the task is to classify a new tuple t in a locally accurate and fair manner. Our framework defines locality relying on a similarity measure s and considers retrieving the k most similar tuples to t in D .

One way to determine the k most similar tuples are kNN algorithms [Bh10]. FALCES uses the kd-tree nearest neighbor approach [Be75] because it is simple and efficient. This method creates a k -dimensional tree that can be precomputed during the offline phase in which the tuples from D are arranged and stored according to the dimensions. During the online phase, when the tuple t is to be classified, the tree can then be searched in $O(\log|D|)$ time.

While searching for the nearest neighbors, we need a similarity metric to identify tuples similar to t . To compare two tuples $t_1 = (x_1, \dots, x_n)$ and $t_2 = (y_1, \dots, y_n)$, FALCES uses the Minkowski-Distance $md(t_1, t_2) = (\sum_{i=1}^n |x_i - y_i|^p)^{\frac{1}{p}}$. It is a generalization of both the Manhattan distance ($p=1$) and the Euclidean distance ($p=2$) and has already proven useful for similar problems such as K-Means algorithms [SYR13].

Using this distance measure, we identify the nearest neighbors of t . However, it must be ensured at this step already that the af metric used in the next step of FALCES receives the necessary information to calculate group fairness. For this, it needs to receive tuples from all groups to be able to produce meaningful results. Therefore, in FALCES, the kNN algorithm is applied to each group separately, which results in $|G|$ trees and $|G| * k$ nearest neighbors for $|G|$ groups.

4.4 Dynamic ensemble selection

Based on the $|G| * k$ tuples defining the local region for a given test sample t , dynamic ensemble selection identifies the ensemble $E = [(m_{c_1}, g_1), \dots, (m_{c_p}, g_p)] \in MC$ that achieves the best local performance. To this end, FALCES follows previous research on dynamic model ensembles [WKB97] and combines these techniques with the af metric. More precisely, using as input MC , we evaluate all model combinations based on af when they classify the $|G| * k$ nearest neighbors of t . The combination E with the lowest af -score is retained.

Example 3. Assume we want to classify a male employee t that is thus considered to be part of g_M . Assuming $k = 20$, kNN retrieves the 20 male and 20 female samples in D most similar to t . The two combinations possible with the classifiers retained after model pruning (see Example 2), i.e., $[(m_1, g_M), (m_4, g_F)]$ and $[(m_2, g_M), (m_5, g_F)]$, are evaluated using the af metric and focusing on the 40 samples of D that form the local region. In our example, let this result in $E = [(m_1, g_M), (m_4, g_F)]$ as this combination reaches the lowest score.

Note that through previous model pruning during the offline phase, the above example needed only to consider 2 instead of 25 classifier combinations. In addition to reducing the number of comparisons, we further reduce the complexity of an individual combination assessment, because the computation of classification predictions for all sets of classifiers and all local $|G| * k$ tuples can be quite time consuming. That is, FALCES precomputes all classification predictions for all tuples in D using all models in M . This allows dynamic ensemble selection to simply look up the necessary predictions instead of repeatedly computing them by applying the classifier for each test sample during the online phase.

4.5 Locally fair classification

Finally, the classifier of the previously identified model ensemble E that achieved best local performance with respect to the af metric and that is associated to the same group as t is used to classify t .

Example 4. Continuing our running example with $E = [(m_1, g_M), (m_4, g_F)]$, m_1 is finally used to classify t , because t belongs to g_M . Considering a different t' of group g_M may

result in a different local region, where for instance $E = [(m_2, g_M), (m_5, g_F)]$ performs better, resulting in the use of m_2 instead.

5 Evaluation

We have implemented the algorithms discussed in Section 4 and present their evaluation in this section. We first describe our experimental setup. We then present and discuss results on combined accuracy and fairness, differences observed for the two *af* metrics, as well as runtime results on the online phase.

5.1 Experimental setup

This section summarizes which different algorithm variants and baseline solutions we compare in our evaluation. We further discuss datasets and metrics we use for benchmarking.

Compared algorithms. We have presented different variants of FALCES, depending on whether or not the training data is split before training and whether or not model pruning is applied. In addition, we compare to the state-of-the-art algorithms. More precisely, we consider the algorithms summarized in Table 1. For the different FALCES variants as well as DCS-LA, which also relies on kNN search, we set $k = 15$.

Datasets and ML models. We use both synthetic and real benchmark data in our evaluation.

We developed a synthetic data generator in order to control different types and degrees of bias in order to study how the different algorithms are affected by these. It generates labeled data for two groups and a binary classification task and allows to control (i) the *group*

Algorithm	Description
FALCES	Our baseline algorithm without splitting before training and without model pruning.
FALCES-SBT	This variant of FALCES splits the dataset for training but does not apply model pruning.
FALCES-PFA	In this variant, model pruning is applied on models trained over the complete training dataset.
FALCES-SBT-PFA	The offline phase performs model pruning on models that have been trained on sub-sets of the training dataset, which has been split according to considered groups.
DCS-LA [WKB97]	A baseline algorithm for dynamic model ensembles that optimizes accuracy, which we extended for FALCES.
Decouple [Dw18]	State-of-the-art algorithm for fair model ensembles, when models are trained using the full training dataset.
Decouple-SBT [Dw18]	Variant of Decouple that trains models on a previously split training dataset.

Tab. 1: Overview of the algorithms compared in our evaluation

Bias type	Parameter settings
Group balance	0.1, 0.2, 0.3, 0.4, 0.5
Social bias	0 , 0.1, 0.2, 0.3, 0.4, 0.5
Implicit bias	0 , 0.1, 0.2, 0.3, 0.4, 0.5

Tab. 2: Different configurations for synthetic data, default values in bold

balance, (ii) the degree of *social bias*, and (iii) *implicit bias*. Group balance describes the percentage group g_1 represents in the full dataset (g_2 implicitly making up the remainder of the dataset), which can be very unbalanced (e.g., only 10% of the training data belongs to g_1) or perfectly balanced at 50%. Social bias refers to bias directly related to the protected attribute defining a group (e.g., gender in our example), reflected by different probabilities for a positive label in the different groups (e.g., women have a lower probability for a positive label than men). Such bias is sometimes also called historical bias, because it reflects direct discrimination of a group in a dataset that commonly labels data based on historical decisions. In our experiments, a social bias of 0 means probabilities are equal for both groups (no discrimination), 0.1 if the probability for g_1 differs by 0.1, and so on. Implicit bias is present in a dataset when, even though groups are not directly discriminated, their label depends on an unfavorable attribute value that occurs more frequently in the protected group, i.e., that is correlated to the protected group. Note that both examples from the press mentioned in the introduction are likely linked to such indirect bias. Similarly to social bias, we vary indirect bias from 0 (none) in increments of 0.1. The generated data in all cases consists of approximately 13,000 tuples. Table 2 summarizes the configurations we used for testing. When not mentioned otherwise, the values are set to the default values highlighted in bold.

We chose the Adult Data Set [DG19], a census income dataset with data from 1994, which is a commonly used dataset in multiple machine learning experiments. This dataset consists of approximately 49,000 tuples and contains various variables, including a binary salary value of yearly income with the threshold of 50K\$, which is our label in the experiments. We chose the attribute “sex” with values “male” and “female” as a sensitive attribute, as well as a combination of the attribute “sex” with the attribute “race” with values “white” and “others”, where we grouped together all other races, because all other races make up $\approx 10\%$ of the dataset.

Each dataset (synthetic and real) is split randomly such that 50% of the dataset serve as training data for model training, 35% for validation to determine emsembles, and 15% for testing the quality of predictions in the online phase.

To get a diverse set of classifiers, we train five different classifiers on our datasets: (i) Decision Tree, (ii) Logistic Regression, (iii) Softmax Regression, (iv) Linear Support Vector Machine, and (v) Nonlinear Support Vector Machine. Given that we have two groups, this results in ten classifiers when we split before training, and five when training on the full dataset.

Metrics. Given that we aim for a good compromise of accuracy and fairness, we use metrics to assess the different algorithms in these two dimensions. We use the well known accuracy-metric commonly used to evaluate machine learning techniques. For fairness, we distinguish between global and local fairness. To study global fairness, we use the “unfairness part” of the metric given by Equation 2 (setting $\lambda = 0$), to which we refer to as global bias (lower values are better). To measure local bias, we define a local region bias metric, which we call *local region discrimination (LRD)*:

$$LRD = \frac{1}{|G| \cdot |D|} \sum_{t_i \in D} \sum_{g_j \in G} \left| \frac{1}{k} \sum_{z_l \in R_{t_i, g_j}} z_l - \frac{1}{k|G|} \sum_{g_m \in G} \sum_{z_l \in R_{t_i, g_m}} z_l \right| \quad (3)$$

where R_{t_i, g_j} is the local data region of t_i comprising the kNN of t_i in group g_j . In this metric the probability of a positive predicted label of each group in the local region is measured against the average probability of a positive predicted label amongst all points in the local region. In this way, the metric reflects the average local fairness.

Using the experimental setup described in this section, we now discuss results we obtained.

5.2 Comparative evaluation in terms of accuracy and fairness

We first present results we obtained when using different algorithms on our synthetic dataset in terms of accuracy, global bias, and local bias.

As a first baseline, we start with a “clean” dataset with no social or implicit bias, and see if changes in group balance have an impact on our three metrics. Essentially, we expect only a marginal effect on accuracy and a low global and local bias, because the input data is a priori unbiased. This is confirmed by the results depicted in Figure 4. Note that instead of plotting absolute accuracy for all methods, we plot the deviation algorithms have in accuracy from the accuracy reached by DCS-LA, reported as percentage points. DCS-LA is not considering bias and optimizes solely for accuracy, which is between 0.76 and 0.79 for DCS-LA over the whole range of considered group balance. The ordinate reporting percentage points, a deviation of -1 means that an algorithm reaches for instance 0.77 instead of 0.78 reached by DCS-LA.

In Figure 4, we observe that all algorithms perform similarly, i.e., for all algorithms, there is some very small fluctuation in accuracy and global bias remains low. For local bias, while being generally low as well, we observe that it steadily increases with increasing imbalance, reaching a relative increase of up to 64% from the balanced case (0.5) to the highest imbalance (at 0.1, where only 10% of the dataset concern one group).

Next, we perform the same analysis again, but this time with an additional social bias of 0.3 introduced to g_1 . The results are summarized in Figure 5. With the introduction of social bias, we observe that deviations in accuracy become more pronounced, in particular for the two variants of the Decouple algorithm. Least affected in terms of accuracy is FALCES-SBT-PFA, actually having comparable or better accuracy than DCS-LA. For both local and global bias, we see that all FALCES variants consistently outperform both Decouple variants and DCS-LA. Also, compared to the previous experiment without social bias, the field has overall shifted upwards. This shows that we cannot fully counter bias originally present in the dataset, but FALCES is best in reducing it while maintaining high accuracy.

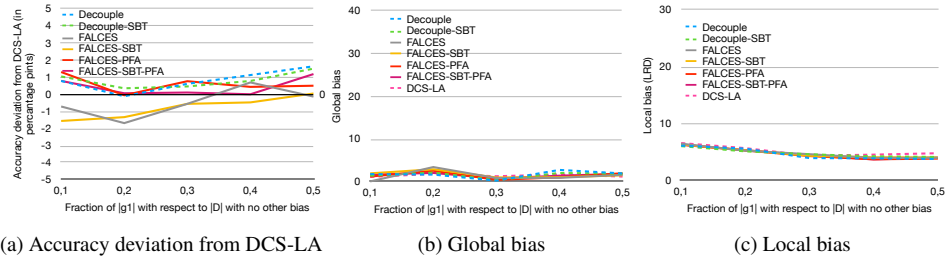


Fig. 4: Results on synthetic data with varying group balance, no social bias, and no implicit bias.

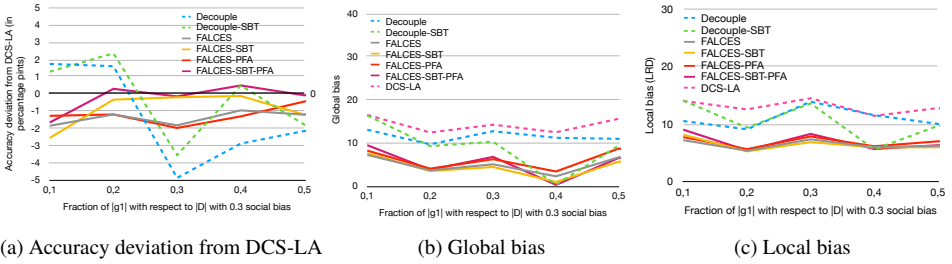


Fig. 5: Results on synthetic data with varying group balance, 0.3 social bias, and no implicit bias.

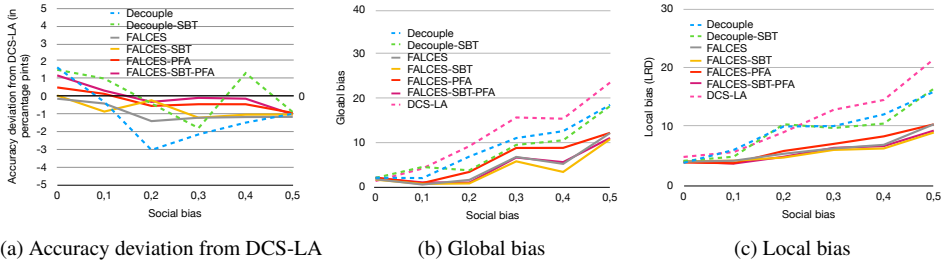


Fig. 6: Results on synthetic data with varying social bias, group balance of 0.5, and no implicit bias.

Our next analysis focuses on the impact different degrees of social bias have on the overall performance, assuming balanced groups without additional implicit bias. Figure 6 reports our results. For accuracy, we observe that all methods fluctuate, but the degradation in accuracy (typically less than 2 percentage points) is tolerable. Our approaches are more robust to social bias than the state-of-the-art Decouple variants, the PFA variants generally being closest to the accuracy reached by DCS-LA. For both local and global bias, a clear upward trend is visible with increasing social bias, showing that the more bias in the input data, the more bias the ensemble generates. However, the gradient of our approaches is less steep and consistently below the baseline methods. This means that the more social bias in the data, the more effective our approaches are in countering the bias to optimize (local) group fairness compared to the state-of-the-art. FALCES-SBT is best in terms of global

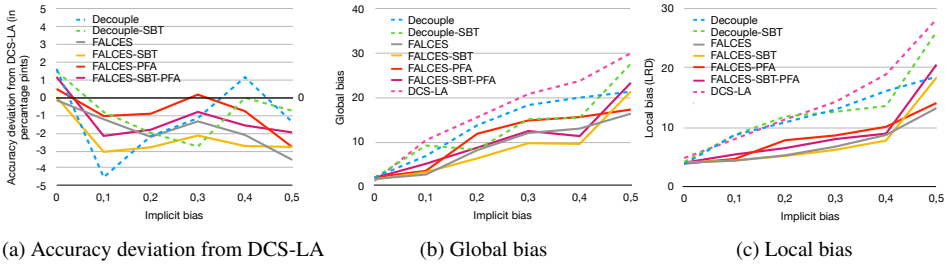


Fig. 7: Results on synthetic data with varying implicit bias, group balance of 0.5, and no social bias.

and local bias, but FALCES-SBT-PFA has similar performance and thus presents a good compromise in datasets with mainly social bias.

We perform a similar analysis for implicit bias in the source data, again assuming a balance of groups (balance = 0.5) and setting social bias to 0. Figure 7 visualizes the results of this set of experiments. Our first observation is that implicit bias impacts all metrics more than the previously considered social bias. As before, in terms of variations in accuracy, these are strongest for the Decouple variants, whereas the PFA variants of our algorithm outperform FALCES and FALCES-SBT. However, looking at both local and global bias, our algorithms without model pruning typically perform better than their counterpart with PFA. The reason for this is that model pruning during the offline phase can prune classifiers that would, during the online phase, be better compared to those retained after model pruning. Nevertheless, in general, our methods outperform the state of the art for a wide range of implicit bias configurations.

We validate our findings on artificial data on the real-world dataset as well. Given that it includes two sensitive attributes (sex and race), we study accuracy, global bias, and local bias when just one attribute is used to form groups (resulting in two groups) and when two attributes are used (resulting in 4 groups). Figure 8 shows results for global and local bias. Results on accuracy confirm that all algorithms perform similarly, it consistently ranges between 0.790 (Decouple) and 0.799 (DCS-LA). As before, we observe that FALCES variants typically are comparable or outperform the three baseline algorithms, both in terms of global and local fairness. With the increasing number of sensitive attributes, we observe that the bias increases for all methods.

In conclusion, we see that our methods improve on the state of the art by offering a better accuracy-fairness compromise than the state of the art Decouple approach (considering global fairness) and the difference in accuracy compared to DCS-LA is typically tolerable. Our methods are also the most robust to different types and degrees of bias (we studied group balance, social bias, and implicit bias). An added benefit is that our methods inherently consider local fairness as well, and our evaluation of local fairness shows that the classifications performed using our algorithms get us closer to equal opportunity for different predefined groups.

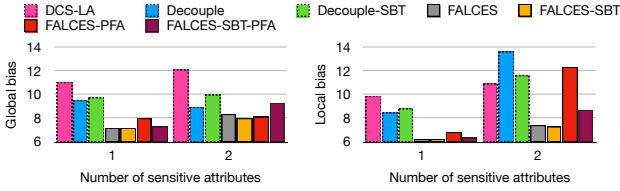
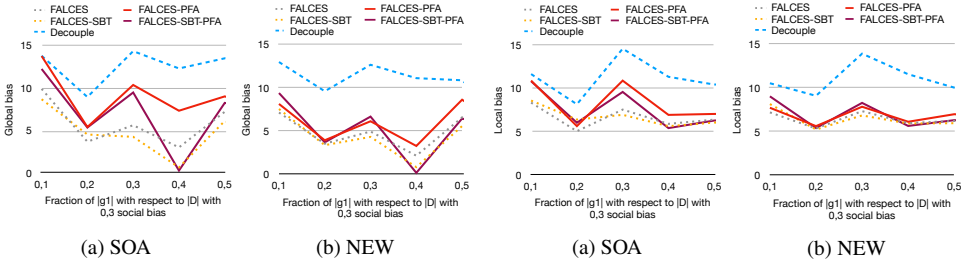


Fig. 8: Global bias (left) and local bias (right) on real-world dataset.

$ G $	FALCES	FALCES-PFA
2	0.58	0.48
4	7.98	1.63

Fig. 9: Average classification time (s) on real-world dataset

Fig. 10: Global bias for varying group balance, 0.3 social bias, and using alternative *af*-metricsFig. 11: Local bias for varying group balance, 0.3 social bias, and alternative *af*-metrics

5.3 Impact of different accuracy-fairness metrics

Section 4.2 has discussed two alternative metrics for *af*, used both during model pruning in the offline phase and dynamic classifier selection in the online phase. All experiments so far have used our extended metric (Equation 2). The next series of experiments investigates how the two options potentially impact the result. We refer to the state-of-the-art metric of Equation 1 as SOA, while our extended metric is labeled NEW. As a reminder, our extension aims at countering the effect on fairness in presence of unbalanced groups. Therefore, we focus our study on evaluating both the global and local bias for different configurations of group balance. As before, accuracy is comparable across all approaches, whether we use SOA or NEW. Figure 10 reports our results on global bias, whereas Figure 11 focuses on local bias. For better readability, we omit the results of Decouple-SBT and DCS-LA, their relative performance to the other approaches being analogous to our previous discussion.

For both global bias and local bias, we see that FALCES variants without model pruning (dotted lines) are comparable when using SOA or NEW. The effect of using a different metric only becomes apparent when model pruning is active. Overall, we see that NEW closes the “bias gap” between FALCES variants with model pruning (solid lines) and those without. This allows our methods to consistently exhibit low bias, especially in comparison to state-of-the-art algorithms like Decouple. This behavior can be explained by the fact the *af* is used by model pruning where group imbalance can cause the pruning of otherwise good classifier combinations. Note that the use of *af* during the online-phase is not sensitive to the choice of the two metrics, because it ensures class balance in the local

region by selecting k members of each group to form a region. Consequently, FALCES and FALCES-SBT are not significantly affected by the choice of metric.

5.4 Runtime evaluation

We also evaluate the efficiency of our approach in its online phase. In particular, we study the effect of model pruning in the offline phase on the online performance. To this end, we run the four variants of FALCES and measure the average runtime to perform online classification for all tuples for which we want a prediction. We report results in Figure 9 on our real-world dataset, on which we can vary the number of groups (either 2 or 4), given two sensitive attributes. In any configuration, we see that model pruning during the offline phase improves the average runtime to classify a test tuple during the online phase. While this improvement is moderate when limiting to two groups, the difference increases as the number of groups increases. This can be explained based on the fact that for two groups and five models trained per group, we have 25 combinations to consider during the online phase when none are previously pruned. This exponentially increases with the number of groups, e.g., for 4 groups, 5^4 combinations need to be tested. Combined with the performance in terms of accuracy and fairness (see Section 5.2), FALCES-SBT-PFA is the method of choice when the number of groups increases.

6 Conclusion

This paper studied the novel problem of making locally fair and accurate classifications to foster equal opportunity decisions. We have presented a general framework to address the problem, as well as FALCES, an implementation of the framework that combines and extends techniques of dynamic model ensembles and fair model ensembles. Our experimental evaluation demonstrated that FALCES generally outperforms the state of the art when it comes to balancing accuracy and fairness for several types and degrees of bias present in the training dataset. Possible avenues for future research include methods that diversify the set of trained models in a controlled way or dynamic and adaptive setting of the parameter k of the kNN search, depending on the density of the data region.

Acknowledgements. Partially supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2120/1 - 390831618.

Bibliography

- [Be75] Bentley, Jon Louis: Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [Bh10] Bhatia, Nitin: Survey of nearest neighbor techniques. *International Journal of Computer Science and Information Security (IJCSIS)*, 8(2):4, 2010.

- [Co20] Coughlan, Sean: Why did the A-level algorithm say no? BBC.com, 2020.
- [CSC18] Cruz, Rafael M.O.; Sabourin, Robert; Cavalcanti, George D.C.: Dynamic classifier selection: Recent advances and perspectives. *Information Fusion*, 41:195–216, 2018.
- [CV10] Calders, Toon; Verwer, Sicco: Three naive Bayes approaches for discrimination-free classification. *Data Mining and Knowledge Discovery*, 21(2):277–292, 2010.
- [Da18] Dastin, Jeffrey: Amazon scraps secret AI recruiting tool that showed bias against women. Reuters.com, 2018.
- [DG19] Dua, Dheeru; Graff, Casey: UCI Machine Learning Repository. 2019.
- [DSM19] Dvornik, Nikita; Schmid, Cordelia; Mairal, Julien: Diversity With Cooperation: Ensemble Methods for Few-Shot Classification. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. pp. 3723 – 3731, 2019.
- [Dw18] Dwork, Cynthia; Immorlica, Nicole; Kalai, Adam Tauman; Leiserson, Max: Decoupled Classifiers for Group-Fair and Efficient Machine Learning. In: *Conference on Fairness, Accountability and Transparency*. volume 81 of *Proceedings of Machine Learning Research*, pp. 119–133, 2018.
- [FSV16] Friedler, Sorelle A.; Scheidegger, Carlos; Venkatasubramanian, Suresh: On the (im)possibility of fairness. *arXiv preprint arXiv:1609.07236*, p. 16, 2016.
- [Ga19] Garg, Sahaj; Perot, Vincent; Limtiaco, Nicole; Taly, Ankur; Chi, Ed H.; Beutel, Alex: Counterfactual Fairness in Text Classification through Robustness. In: *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*. AIES '19. ACM, p. 219–226, 2019.
- [KC09] Kamiran, Faisal; Calders, Toon: Classifying without discriminating. In: *2009 2nd International Conference on Computer, Control and Communication*. IEEE, pp. 1–6, 2009.
- [Po06] Polikar, Robi: Ensemble Based Systems in Decision Making. *IEEE Circuits and Systems Magazine*, 6(3):21–45, 2006.
- [PRT08] Pedreschi, Dino; Ruggieri, Salvatore; Turini, Franco: Discrimination-aware Data Mining. In: *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD '08. ACM Press, pp. 560–568, 2008.
- [SHX19] Shen, Zhiqiang; He, Zhankui; Xue, Xiangyang: MEAL: Multi-Model Ensemble via Adversarial Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:4886–4893, 2019.
- [SYR13] Singh, Archana; Yadav, Avantika; Rana, Ajay: K-means with Three different Distance Metrics. *International Journal of Computer Applications*, 67(10):13–17, 2013.
- [WKB97] Woods, Kevin; Kegelmeyer, W. Philip; Bowyer, Kevin: Combination of multiple classifiers using local accuracy estimates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):405–410, 1997.
- [Zh19] Zheng, Hao; Zhang, Yizhe; Yang, Lin; Liang, Peixian; Zhao, Zhuo; Wang, Chaoli; Chen, Danny Z.: A New Ensemble Learning Framework for 3D Biomedical Image Segmentation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:5909–5916, 2019.
- [Žl17] Žliobaitė, Indrė: Measuring discrimination in algorithmic decision making. *Data Mining and Knowledge Discovery*, 31(4):1060–1089, 2017.

Cluster Flow — an Advanced Concept for Ensemble-Enabling, Interactive Clustering

Sandra Obermeier,¹ Anna Beer,¹ Florian Wahl,¹ Thomas Seidl¹

Abstract: Even though most clustering algorithms serve knowledge discovery in fields other than computer science, most of them still require users to be familiar with programming or data mining to some extent. As that often prevents efficient research, we developed an easy to use, highly explainable clustering method accompanied by an interactive tool for clustering. It is based on intuitively understandable kNN graphs and the subsequent application of adaptable filters, which can be combined ensemble-like and iteratively and prune unnecessary or misleading edges. For a first overview of the data, fully automatic predefined filter cascades deliver robust results. A selection of simple filters and combination methods that can be chosen interactively yield very good results on benchmark datasets compared to various algorithms.

Keywords: Clustering; Interactive; kNN; Ensemble; Explainability

1 Introduction

Researchers in virtually all areas can benefit from clustering their data at some point. From natural sciences over social studies to economics — data is gathered everywhere. Clustering provides many advantages: while the main goal is to find groups of similar objects, it can also help gather valuable hidden information from the data or identify essential attributes. While researchers are experts in their field, they often do not have sufficient background knowledge about clustering methods and, for the sake of simplicity, use old traditional algorithms that may not even fit their data.

As datasets from different research areas contain different types of clusters, ensemble methods proved themselves as suitable for users without profound knowledge in data science. Nevertheless, ensemble methods can be even less understandable “black boxes” than only one algorithm, as they combine different clustering algorithms. Interactive and visual approaches offer great possibilities to make these black boxes more accessible and embody a good solution for the desired balance: To make the powerful tool of clustering accessible to researchers from all fields so that they can create the most meaningful and transparent clusterings with as little effort and background knowledge as possible.

¹ LMU Munich, Institut für Informatik, Oettingenstr. 67, 80538 München, Germany, [obermeier,beer,seidl]@dbs.ifi.lmu.de

A high explainability is thus equally important as public availability. Due to the lack of those in current methods, surprisingly, many researchers still group their data manually, which is disastrous regarding the reproducibility of results and the whole research's objectivity.

Especially complex clustering methods such as ensemble methods are hard to visualize, and accordingly, it is tough to create interaction possibilities on intermediate levels that allow the user to intervene, guide and better understand the process.

To solve these problems, we developed a concept with a prototypical implementation called Cluster Flow. It combines kNN-based approaches for clustering on graphs with a modular and easy to understand architecture. It is simple but simultaneously provides enough flexibility to accomplish difficult clustering tasks. We publish our code at <https://github.com/sobermeier/cluster-flow>. Cluster Flow combines the advantages of ensemble clustering with interactive clustering: users of all areas can easily apply and compose various intuitively understandable cluster improvement steps iteratively to explore and cluster their data. Our method, which we describe in detail in Section 3, is based on kNN graphs as they represent one of the best foundations for clustering and offer several advantages: they are highly explainable, suitable for anytime changes, and interactive approaches, and they can enable finding non-convex shaped clusters as well as clusters of different densities. We developed multiple intuitively understandable filter methods partially based on existing methods to prune edges connecting clusters. They can be combined sequentially or in parallel (ensemble-like). Users can fine-tune parameters, change filters, and explore the dataset at any time, guided by a well-structured prototypical user interface as explained in Section 4. Extensive experiments in Section 5 show that our concept, applying pruning-strategies on kNN graphs, achieves better clusterings than several other algorithms with their best parameter settings. Simultaneously, the method is robust and suitable for exploring data: with fixed parameters for all tested datasets, our fully automatic predefined filter cascades yield better results than comparative methods. Our main contributions can be summarized as follows.

- We propose Cluster Flow, an advanced clustering concept based on kNN graphs by deleting edges through filters.
- Due to its well-thought-out modular design, interactions can be easily integrated on intermediate stages while also providing step-by-step visualizations of the intermediate clustering results.
- Even beyond this, we provide predefined filter cascades that achieve competitive results fully automatically.
- A prototypical implementation serves as a proof-of-concept and demonstrates the power of our proposed approach.

2 Related Work

We address here the three main components that comprise our Cluster Flow concept. As most filter methods are based on diverse graphs describing the data, and we need to evaluate our clustering results objectively, we introduce some graph-based methods in Subsection 2.1, which we also use as a baseline for our experiments in Section 5. Combining the filters can be seen as an ensemble approach; thus, we introduce the most relevant ensemble-based clustering methods in Subsection 2.2. Subsection 2.3 concludes this section by setting our concept into context regarding existing interactive approaches.

2.1 Graph-Based Clustering

Clustering algorithms can rely on different graphs extracted from the original data, e.g., ϵ -range graphs or diverse variants of kNN graphs, where we focus on the latter. Some approaches rely on a mutual kNN graph (MkNN, see Section 3.1). Existing works include taking the plain MkNN graph, where a connected component with two or more points form a cluster and otherwise are considered outliers [Br97], or slightly advanced ones where a weighted MkNN graph is used to capture clique-like structures [SB14]. Choosing the optimal value for k is especially difficult for MkNN, which are inherently sparser than, e.g., symmetric kNN graphs (see Section 3.1).

The hierarchical clustering algorithm *CHAMELEON* [KHK99] is based on symmetric kNN graphs and consists of two phases. First, the kNN graph is partitioned into small sub-clusters by repeatedly splitting the currently largest sub-cluster, such that the edge cut is minimized until the largest sub-cluster contains fewer nodes than a user-given parameter *MinSize*. Secondly, these sub-clusters are recombined using an agglomerative hierarchical clustering algorithm concerning their relative inter-connectivity and closeness. The merging algorithm terminates when only one cluster remains, or no pair of clusters satisfies the condition of having high enough relative inter-connectivity and relative closeness.

Girvan-Newman Algorithm [GN02] is an approach for detecting community structures in graphs. The authors introduce a measure called *edge betweenness* which corresponds to the number of shortest paths that run along this edge. All shortest paths between nodes of different communities go along at least one edge that connects the communities. Thus, the *edge betweenness* score of such an inter-community edge is higher. The algorithm iteratively removes the edge with the highest *edge betweenness* and recalculates it for the remaining edges until there are no more edges in the graph. The result of the algorithm is a dendrogram that reveals the community structure of the underlying graph. However, this method has a relatively high runtime with $O(m^2 \cdot n)$ for graphs with m edges and n nodes.

Spectral Clustering [SM00] is based on a similarity graph and its weighted adjacency matrix, the first k eigenvectors are calculated. A kNN graph can be used as a similarity graph with distances between points as weights. Clustering is then performed with k-means

on the matrix's rows, which contains the previously calculated eigenvectors as columns. The resulting clustering assignment of k-means corresponds to the clustering alignment of the original points. Spectral clustering also inherits the disadvantages from the additional partitioning step, such as k-means to a certain degree.

2.2 Ensemble based Clustering

Our approach is closely related to ensemble clustering methods since our filter strategy allows combinations to find a consensus. Cluster ensembles, sometimes also referred to as clustering aggregations or consensus clustering, combine several cluster algorithms to obtain a single result of better quality than each cluster individually. Usually, they are based on two steps, namely the *generation*, where different partitions are obtained, and the *consensus*, where these partitions are integrated into one final partition. Ensemble methods should at least meet the following four criteria [VPRS11]:

- **Robustness:** The average performance must be better than the single clustering algorithms.
- **Consistency:** The combined result should be very similar to all combined single clustering algorithm results.
- **Novelty:** The ensemble must allow finding solutions unattainable by single clustering algorithms.
- **Stability:** The results must be less sensitive to noise and outliers.

However, according to the same authors, identifying the best result is hard, but the general idea behind ensemble methods is that several algorithms' combined decisions should be more reliable than any individual one. Several existing works focus on the clustering techniques [Li15, FJ05, Wu13] while others focus on finding the right consensus [SG02], and allow for using different clustering methods.

However, in these approaches, determining the consensus functions is only applicable for experts and integrating different techniques is even more complex and challenging to understand to scientists from other domains. In contrast to that, our edge-deletion concept allows for a smooth integration of establishing consensus across filters while at the same time the intermediate results are always visible and understandable. Our approach differs from existing approaches since it is possible to apply the ensemble clustering paradigm as an intermediate step. The user can access the result at a fine-granular level and directly see how different filters agree upon activities to identify crucial spots or steer the end product into a more conservative or progressive direction. These advantages come because our approach combines the power of ensemble clustering and interactive clustering. We discuss the latter in the next section.

2.3 Interactive Clustering

The overall goal of interactive clustering is to engage the user as far as possible in the clustering process, not only to allow the user to make the result fit their preferences but also to make it understandable. In our context, we utterly differentiate from methods that solely enable visual exploration of the result and are only considering methods that allow interaction within the algorithmic loop. [Ba20] provide a thorough survey of interactive clustering. Over 100 papers related to interactive clustering are analyzed regarding the stage and type of interaction, user feedback, evaluation criteria, data, and clustering methods. The authors distinguish three groups of stages in which interaction occurs. (1) Interaction on clustering results, (2) interaction on model/algorithm level, and (3) machine-initiated interaction. Our concept belongs to the second group since the user's interactions directly happen at the algorithm level by tweaking parameters rather than at the clustering results. We evaluate the clustering result on an objective basis rather than conducting a user study for subjective evaluation. A user study might help develop an appealing and intuitive graphical user interface but is not within this work scope. Also, visualization methods for supporting interactive ensemble clustering like AUGUR [HHL10] could be incorporated for future work.

To conclude this section and put our work into the context of related works, our concept combines three main components that fit together enormously well. First, kNN-graphs are inherently well understandable for humans. Second, based on the kNN-graph, filter strategies are applied. The filters focus on different hidden structures in the data, but all result in the same action, namely deleting edges. Therefore, finding a consensus among them in an ensemble-like manner does not require complex mathematical functions but rather comparison on edge level and can thus be well visualized and understood. Third, the modular design allows interaction on each stage and the continuous visualization of intermediate results.

3 Cluster Flow

Cluster Flow works on a kNN graph of the input, for which we present multiple options in Section 3.1. Elaborated filters, which we introduce in Section 3.2, delete edges between clusters, and thus the graph decomposes into several smaller graphs representing one cluster each.

3.1 Build kNN Graphs

Cluster Flow allows to choose between different variants of the kNN graph, as they can have a severe impact on the clustering result [MLH09]: basic kNN graphs, symmetric kNN graphs, and mutual kNN graphs.

Within this paper, we use the following notation: Let U be an arbitrary set and $DB \subseteq U$, $|U| < \infty$ be a dataset. A **kNN** graph consists of nodes corresponding to the points of a dataset and directed edges from every node to its k nearest neighbors (kNN). The kNN graph is a directed graph where an edge (p_i, p_j) from point p_i to point p_j exists if and only if p_j belongs to the k -neighborhood of p_i [HKF04]:

$$edges = \{(p_i, p_j) \mid \forall p_i, p_j \in DB: i \neq j \wedge p_j \in kNN(p_i)\} \quad (1)$$

A **symmetric kNN** graph has a higher connectivity than the kNN graph: it is an undirected graph where an edge (p_i, p_j) from point p_i to point p_j exists if p_j is part of k -neighborhood of p_i or vice versa [HKF04, MHVL07]:

$$edges = \{(p_i, p_j) \mid \forall p_i, p_j \in DB: i \neq j \wedge (p_j \in kNN(p_i) \vee p_i \in kNN(p_j))\}. \quad (2)$$

A **Mutual k-Nearest Neighbor (MkNN)** graph is an undirected graph, where edges exist between two points p_i and p_j if both points belong to each other's k -neighborhood [Br97, HKF04]:

$$edges = \{(p_i, p_j) \mid \forall p_i, p_j \in DB: i \neq j \wedge p_i \in kNN(p_j) \wedge p_j \in kNN(p_i)\}. \quad (3)$$

The **RkNN** graph connects points to their reverse nearest neighbors, i.e., its adjacency matrix is the transpose of the adjacency matrix of the corresponding kNN graph. A point p is a reverse nearest neighbor of a point q , iff q is a nearest neighbor of p .

As we later only delete edges and never add edges, points of a cluster must have a connection in the graph. Thus, a symmetric kNN graph with its inherent rather high connectivity is usually suitable for our approach. If users are interested in the most significant clusters or the core points of a cluster, an MkNN graph can be a good choice [MHVL07]. Note that k for MkNN graphs should be higher than for symmetric kNN graphs to ensure a certain degree of connectivity. Even though we use unweighted graphs for all filters, we save and reuse the distances calculated in this step if needed.

3.2 Filters

In the following, we introduce filters that can be applied to the kNN graph. Filters are easily accessible for users of diverse domains and delete different edges depending on the graph's properties, as we illustrate with selected example graphs shown in Figure 1.

3.2.1 Edge-Distance Filter (EDF)

Since points that are close to each other and connected by a short edge in the kNN graph are likely to be in the same cluster, EDF deletes edges longer than a threshold t ,

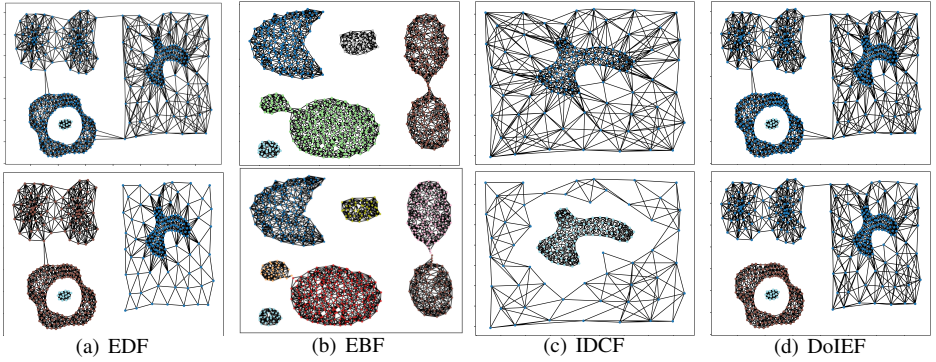


Fig. 1: Application of the Filters. Top: Symmetric kNN graph with $k=10$ of the original datasets. Bottom: After filter application. Each connected component, i.e. cluster, is indicated by a different color.

granting certain reachability for the clusters. The filter considers each connected component individually, which has two advantages: (1) We can run the filter in parallel on several connected components to save computation time, and (2) t is not global but can be chosen for each connected component individually, depending on its edges. This enables maintaining connected components with different densities, which will later result in clusters. The threshold t depends on the mean μ and standard deviation σ of the edge distances in a connected component, on which the filter is applied, where σ is weighted by a parameter $p \in \mathbf{R}$: $t = \mu + p \cdot \sigma$. An empirically good value for p is between 1 and 3. Figure 1(a) shows an example application of this filter on the Compound dataset. The top image displays the symmetric kNN graph with $k=10$ on the original dataset, which is the input for this filter. The bottom figure displays the result after applying EDF with $p=2$ where several unwanted edges have been removed correctly. The filter runs with a complexity of $O(m)$ where m is the number of edges. Note that the distances between all points can be reused from the kNN graph generation.

3.2.2 Edge-Betweenness Filter (EBF)

This filter is based on the Girvan-Newman algorithm and uses the edge betweenness measure to identify and reduce inter-community connections. It works on the assumption that loosely connected components belong to different clusters. This filter iteratively removes the edges with the highest edge betweenness, where i is the number of iterations and p is the number of edges to delete. As smaller clusters have fewer paths connecting all nodes than larger clusters, misclassification, i.e., deleting wrong edges, has a higher impact on them. To overcome this, we make our filter more restrictive, i.e., we scale the number of removed

edges per iteration by the total number of edges $|edges|$ within a connected component. We also constrain that in each iteration, a minimum of one edge is deleted. Parameter r is then defined by: $r = \max(|edges| \cdot p, 1)$.

Figure 1(b) shows an example application of the edge betweenness filter on the Aggregation dataset. The input graph on top is generated with $k = 10$. We set the filter parameters as follows: $i = 5$; i.e., five iterations were performed, and $p = 0.0075$; i.e., 0.75% of the edges were removed from each connected component in each round. The bottom figure shows the result after all iterations. The inter-community connections have been detected correctly, and the initial five connected components have been divided into seven, which fit the ground truth clusters and are highlighted by different colors.

The original Girvan-Newman algorithm has a worst-case complexity of $O(m^2 \cdot n)$, where n is the number of nodes and m is the number of edges. In our modified setting, we can decrease this complexity. First, instead of performing a full hierarchical clustering down to every node, we only run the algorithm for i iterations to identify the top inter-community connections, where $i \ll m$. Secondly, originally only one edge is removed in each iteration. In our case, we increase the number of edges that are deleted in each round to the value of a parameter r . This way, the user can decide how precise the final result should be. Since the complexity for one round of the original algorithm is $O(m \cdot n)$, the filter's application with i rounds has the complexity of $O(m \cdot n \cdot i)$. It is advisable not to apply the filter in the beginning but rather when the complete dataset is already segmented in several connected components to reduce the complexity (For example, by first applying a filter with lower complexity, e.g., the *EDF*).

3.2.3 Inter-Density Connection Filter (IDCF)

This filter assumes that two points located in regions of different densities also belong to two different clusters. The sparseness of the neighborhood of a point is defined by the average distance to its k -nearest-neighbors, which equals to the sparseness estimation presented in [SRS00] and is used for outlier detection. We call an edge between two nodes with very different dense neighborhoods **inter-density connection**. The inter-density connection filter aims at detecting these edges to classify them as unwanted. The **density difference of an edge** is defined as the absolute difference of the sparseness of the two nodes it connects. If this density difference is higher than a threshold t , the edge is classified as unwanted. Given μ as the average of the density difference of all edges, σ as its standard derivation and p as a user-defined parameter to regulate the sensitivity of the filter, the threshold t is defined as: $t = \mu + p \cdot \sigma$. Again, the filter is applied to each connected component separately. The sparseness estimation is made on the basis of the directed kNN graph, but only the edges which also exist in the current state of the undirected graph will be considered so that each point has 0 to k considered nearest neighbors. We need the directed graph for this filter, since edges to outliers, which more likely exist in an undirected graph where

the outlier only needs to belong either to the kNN *or* the reverse kNN set, may lead to misclassification. A connected outlier forces an increase in the sparseness estimation for the whole connected component and may lead to the deletion of edges between actual cluster nodes. Nevertheless, we do not have to recompute all kNN relationships since we have created and saved a directed kNN graph at the graph generation step that contains all kNN relationships and distances. Figure 1(c) shows an example application of the inter-density connection filter on a sample from the Compound dataset. On the top, the original input graph is depicted (symmetric kNN graph with $k = 10$). On the bottom, the resulting graph after applying the filter two times with $p = 1.5$ is shown. The filter can separate the inner, more dense cluster from the outer data points.

3.2.4 Distance of Incoming Edges Filter (DoIEF)

Similar to the *EDF*, this filter considers the length of edges. However, unlike the aforementioned, this filter focuses on each node separately and uses the directed kNN graph. The filter considers all incoming edges of a node and therefore requires a calculation of the reverse kNN relationships based on the directed kNN graph in advance. We classify the incoming edges of a node, i.e., edges connecting nodes of the reverse kNN set of the node, as unwanted if their length exceeds a threshold t . The threshold t is defined as: $t = \mu + p \cdot \sigma$, where μ is the average edge length, and σ is the standard derivation of the edge lengths. This filter is used for separating outliers or boundary nodes from other connected components. The intuition behind this filter is that nodes with long edges to their k-nearest neighbor are likely to be outliers or at least probably not part of the cluster to which that neighbor is currently connected. We use the incoming edges for this filter because this is more restrictive than using the outgoing edges. The RkNN and the kNN relationship are not symmetric, i.e., every node has the same number of outgoing edges, but not every node has incoming edges. If we took the outgoing edges, every node in the whole graph would be considered, including those that do not belong to the kNN set of any other node. However, by taking the incoming edges, we limit the considered nodes to those that are part of the kNN set of at least one other node and thus decrease the computational costs and the probability of deleting wanted edges. Figure 1(d) shows a symmetric kNN graph as input on the top, which was generated with $k = 10$ on the Compound dataset. The result after the first application of the *DoIEF* with $p = 2$ is shown on the bottom. The illustration shows that the *DoIEF* deletes edges that connect boundary points of different clusters.

3.3 Combination of Filter Results

We present two different filter strategies: using filters *sequentially* and using filters in *parallel*. The former applies filters consecutively, using results from the previous filter as input for the next filter. Filters can be applied multiple times, and different types of filters can be concatenated. As filters are applied on each connected component separately, an

iterative application of the same filter often leads to better results than a one-time application with relatively high respectively low threshold values. In the beginning, there might be only one component connecting all points. Iterative filters can consider different cluster structures, and the individual clusters can develop selectively. The second strategy embodies the classical cluster ensemble approach's central idea, where different clustering algorithms are performed on the same input, and afterward, a consensus between the different results is determined. We adopt this idea for our concept by applying different filters in parallel on the same input graph. Each filter determines independently which edges should be deleted. Afterward, a common consensus is determined, which can be chosen conservatively, e.g., all filters must agree to the deletion of an edge to ensure a safe deletion within the final result, or progressively, e.g., only 50% of filters need to propose the deletion of an edge to result in a deletion within the final result. Cluster Flow is designed so that the two strategies can be easily combined.

3.4 Concept Overview and Discussion

Figure 2 abstracts a possible manifestation of the Cluster Flow architecture. A gray box represents an atomic building block (either the initial graph generation step, one of the proposed filters, or a consensus component). Important is the graph construction at the beginning, which is the basis for the further procedure. After this mandatory first step, the user can apply any filter presented in the previous paragraphs. Interaction possibilities to tweak parameters and visualizations of intermediate results are integrated at each building block and enable maintaining the overview at all times. The red, solid framed box indicates a sequential filter chain, while the blue dotted framed box shows a parallel filter strategy. A filter always works on the result of the previous building block. Each filter in the parallel component works on the input from the previous filter independently. Finding a consensus is also an independent building block that determines how many parallel filters need to agree upon deleting an edge to delete an edge ultimately. The output of the consensus-building block results in a subsequent filter's input, if one is applied, or in this case, as the final result. We also want to emphasize that the modular design allows storing intermediate results that do not require recalculation whenever a subsequent filter is updated. With this, it is possible to try out different parameters without starting from the bottom. The user experience greatly benefits from this architecture. The experiments were not runtime optimized, but a single filter's execution is in the millisecond to second range for the tested datasets. A significant advantage of our concept is its great flexibility, but in some sense, it might also be challenging to find a good way to start. We recommend relying on a filter-refinement-like strategy to start with a fast, non-exact deletion of edges and go over to more costly fine-tuning. The *EDF* is the most simple filter in our repertoire and is also runtime-efficient. Though it does not delete edges between, e.g., clusters of different densities like *IDCF*, it is a perfect start to delete many unnecessary edges without complex computations, allowing additional filters to work on a fraction of the original edges. The *EBF* is the most complicated filter, which is well suited to be applied at the end as a refinement step. The combination of filters is

a good strategy for either the start or the end, depending on the goal of the clustering. If a conservative approach is generally preferred, the combination of filters can be applied initially, which, depending on the consensus function, deletes only obviously unwanted edges to prevent premature deletion of edges. However, the combination is also suitable to serve as a refinement step.

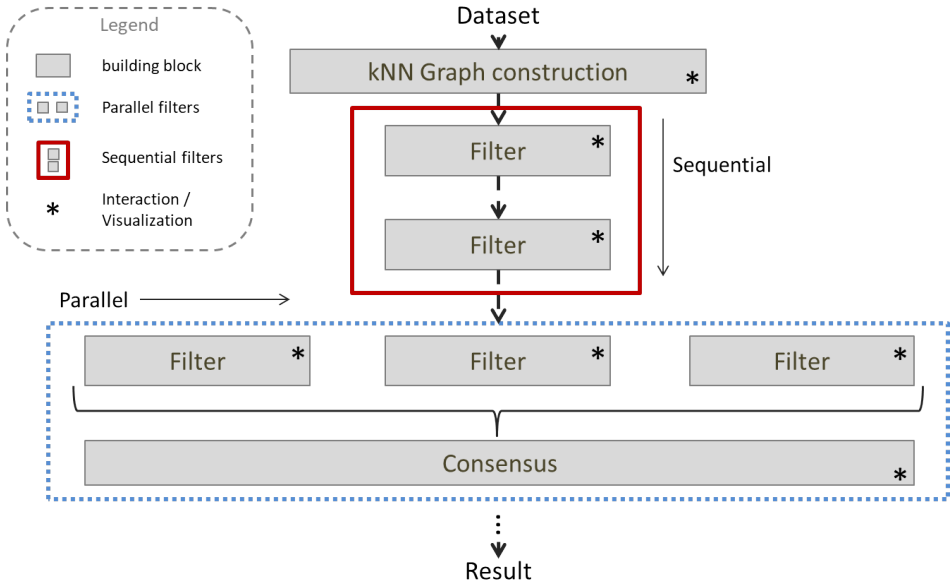


Fig. 2: Design concept of Cluster Flow.

4 Interactive Clustering with Cluster Flow

Due to the modular character and the step-by-step application of the filters, Cluster Flow is well suited for interactive clustering. As a proof of concept, we implemented a lightweight tool that provides a simple interface for loading datasets in CSV format, creating and saving individual projects. Within each project, it is possible to add filters, either at each step for the sequential case or several filters included in one step for the parallel case. One step is represented by a tile containing one or more inner cards, which offer a visualization of the current result on the left and configuration options on the right. For simplicity, each card offers the option to visualize the current result graph within a 2D view, a 3D view, or to apply PCA decomposition [Pe01] for dimensionality reduction. The first step within a project is always the kNN graph generation. Here users can choose the graph type, i.e., symmetric or mutual, and the value for k . Users can attach filters to form a chain in which each filter is executed one after another. Each filter step can be executed and adjusted

Tab. 1: Evaluation datasets with number of clusters c and number of dimensions d .

Set 1 (Gaussian)				Set 2 (Non-Convex)				Set 3 (Mixture)			
Name	c	d	Size	Name	c	d	Size	Name	c	d	Size
Cassini ³	3	2	1000	Two Moons ³	2	2	300	Aggregation[GMT07] ⁴	7	2	788
Cuboids ³	4	3	1002	Donutcurves ³	4	2	1000	Compound[Za71] ⁴	6	2	399
Hypercube ³	8	3	800	Long2 ³	2	2	1000	Pathbased[CY08] ⁴	3	2	300
Cure-t0-2000n-2D ³	3	2	2000	Dartboard1 ³	4	2	1000	Lsun ³	3	2	400
Pmf ³	5	3	649	Donut3 ³	3	2	999	Spiralsquare ³	6	2	1500
Twenty ³	20	2	1000	Smile2 ³	4	2	1000	Longsquare ³	6	2	900
Twodiamonds ³	2	2	800	Zelnik1 ³	3	2	299	Dpc ³	6	2	1000
Spherical_4_3 ³	4	3	400	Zelnik5 ³	4	2	512	Target ³	6	2	770
Zelnik4 ³	5	2	622	Jain[JL05] ⁴	2	2	373	R1_complete ⁵	4	2	600
R15[VRB02] ⁴	15	2	600	Spiral[CY08] ⁴	3	2	312	Mouse ⁶	4	2	500

separately, allowing a high degree of transparency and intervention. However, if a previous filter parameter has been changed so that the resulting kNN graph changes, all subsequent filters are recalculated because their input has changed. A traffic-light system indicates the status of computations. The tool is a local web application written in Python, so it is platform-independent. We used Flask¹ as web framework and SQLite² for the database. Figure 3 shows an example screenshot of the interactive tool. Our prototype can reuse already computed values when chaining filters as far as possible to prevent the calculation costs from increasing proportionally per added filter and ensure a pleasant, smooth usage. It fulfills all criteria of [VPRS11] explained in Section 2: (1) robustness, (2) consistency, (3) novelty, and (4) stability. (1) Cluster Flow is robust, i.e., the ensemble is on average better than its single components. Especially when looking at complex datasets with diverse types of clusters, the superiority of combining several filters becomes obvious. (2) Since Cluster Flow’s consensus strategies are simple operations on sets or majority votes, results are comprehensibly similar to their components’ results. (3) The combination of filters produces novel results, which cannot be achieved by a single filter since there are datasets for which one filter cannot delete all necessary edges for correct clustering, even though another one could. E.g., datasets containing clusters of different densities and clusters connected by a chain. Combining both can lead to a perfect result. (4) Using an adequate consensus strategy reduces the sensitivity regarding noise and outliers.

¹ <https://palletsprojects.com/p/flask/>

² <https://www.sqlite.org/index.html>

³ <https://github.com/deric/clustering-benchmark/tree/master/src/main/resources/datasets/artificial>

⁴ <http://cs.joensuu.fi/sipu/datasets/>

⁵ <https://github.com/wahlflo/Datasets>

⁶ <https://elki-project.github.io/datasets/>

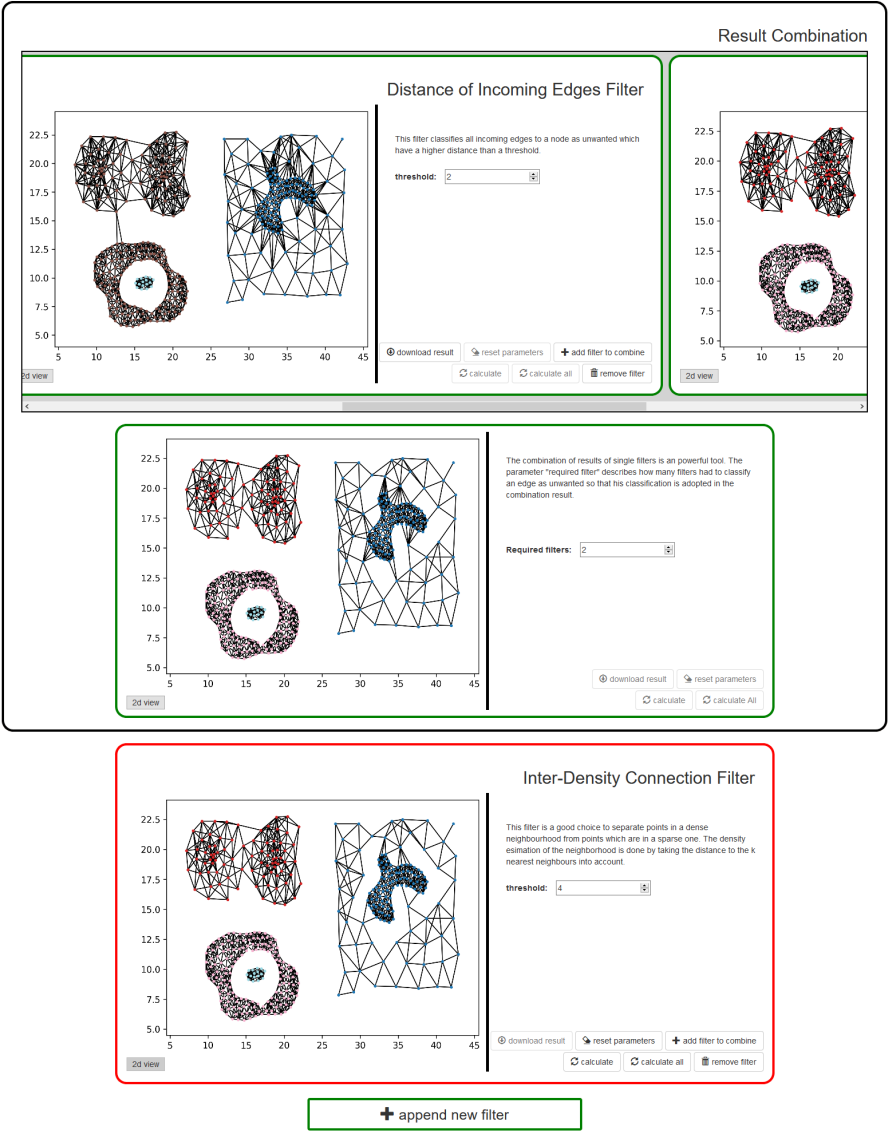


Fig. 3: Screenshot of the prototypical interactive clustering tool. The top row contains parallel filters where the green frame indicates that the calculation has been finished. The bottom row shows a single sequential filter, which is still working.

Tab. 2: Parameter settings used for Cluster Flow.

Parameter	Description	Range	PFC1	PFC2
<i>graph_type</i>	Type of the generated graph.	[symmetric, mutual]	symmetric	symmetric
<i>k</i>	Number of nearest neighbors.	[10, 12, 14]	14	14
<i>EDF_p</i>	Parameter <i>p</i> of the <i>EDF</i> .	[1.5, 2, 2.5, 3]	3	2.5
<i>DoIEF_p</i>	Parameter <i>p</i> of the <i>DoIEF</i> .	[1.5, 2, 2.5, 3]	-	1.5
<i>IDCF_p</i>	Parameter <i>p</i> of the <i>IDCF</i> .	[2, 3, 4, 5]	-	5
<i>EBF_p</i>	Parameter <i>p</i> of the <i>EBF</i> .	[0.0025, 0.005, 0.0075]	0.0075	0.0075
<i>EBF_i</i>	Number of iterations of the <i>EBF</i> .	[0, 1, 2, 3, 5, 7, 9]	7	7
<i>Consensus_n</i>	Number of filters to agree upon deletion.	[2, 3]	-	2

5 Experiments

5.1 Datasets

We evaluate Cluster Flow on 30 publicly available clustering benchmark datasets as described in Table 1. We grouped them into three groups based on the type of clusters they contain: In the first set, data sets contain Gaussian-like clusters, in the second set, they contain non-convex clusters, and in the third set, they contain a mixture of different types. Additionally, we evaluate on several high dimensional datasets taken from [FS18] and first introduced by [FVH06]. These all have 16 clusters and 1024 points, their dimensionality is $d \in [32, 64, 128, 256, 512]$, and they are called dim032, dim064, dim128, etc.

5.2 Baseline

We want to evaluate our approach on an objective basis. For this, we compare our concept with relevant graph-based methods and other established clustering methods. More precisely, we evaluated using the following methods and performed a grid search on the corresponding parameter settings:

- k-means [L182]: $k_{k-means}: \{c - 2, \dots, c + 2\}^7$
- CHAMELEON: $k: [5, 10, 15]$, $MinSize: [2\%, 3\%]$, $\alpha: [1.5, 2.0, 2.5]$
- MkNN clustering: $k: \{3, 4, \dots, 20\}$
- Rock [BKS19]: $tmax: [10, 15, 20]$
- DBSCAN [Es96]: $MinPoints: \{2, 3, \dots, 15\}$, $\epsilon: \{0.01, 0.02, \dots, 0.4\}$
- Spectral clustering [VL07]: $k_{knn}: \{10, 15\}$, $k_{k-means}: \{c - 2, \dots, c + 2\}^7$

⁷ c stands for the number of ground truth clusters

5.3 (Predefined) Filter Cascades

An essential characteristic of Cluster Flow is its high level of flexibility. To ensure better reproducibility and comparability, we here describe filter cascades, which consist of a defined filter composition. The first filter cascade (*FC1*) solely relies on the *sequential* strategy and incorporates the *EDF* and the *EBF*. The *EDF* is applied repeatedly until no edges are newly classified as unwanted. After that, the *EBF* is applied multiple times.

The second filter cascade (*FC2*) relies on a combination of the *sequential* and the *parallel* strategy. The *EDF*, *DoIEF* and the *IDCF* are applied in parallel on the same input dataset. An edge is classified as unwanted if, at minimum, two of the three filters classified it as such. Afterward, the *EBF* filter is applied multiple times. To make our concept as simple as possible and to obviate time-consuming parameter searches, we also tested both filter cascades with constant, predefined parameters over differently structured data sets, i.e., in a fully automatic setting without user interaction. Table 2 summarizes the tested parameters and their ranges in general as well as the fixed hyper-parameters for the predefined filter cascades (*PFC1*, *PFC2*) that were used in the subsequent analysis. Figure 4 shows the construction of *PFC1* and *PFC2* for a better understanding. These two filter cascades follow different goals. *PFC1* is a more progressive approach that tries to remove many edges directly from the beginning. The *EDF* is a good choice for this, as the focus is solely on the distance between two edges without considering the neighborhood. It is also one of the simplest and fastest filters we propose and thus serves as a good first filter to delete the most obvious edges. The *EBF* is more complex but also more powerful since it can detect bridges between communities. This filter takes more runtime than the other filters, and we recommend applying it towards the end where a refinement is needed. *FC1* could also be seen as a filter-refinement procedure, where the *EDF* deletes the most obvious edges fast, and the *EBF* fine-tunes the result. *PFC2*, in contrast, is more conservatively constructed; that is, in case of doubt, an edge is rather not deleted so as not to cause clusters to decay prematurely. The parallel building block in the beginning only deletes an edge if the majority of the three filters agrees upon it to give a more reliable result. The powerful *EBF* is then used again for fine-tuning the result. In terms of objective evaluation, *PFC1* often performs better, but for sensitive applications where the dataset must not be split up in too many clusters too fast, *PFC2* is a good option.

5.4 Performance with varying parameters

The left part of Table 3 shows the average F1-scores of all evaluated clustering algorithms for each of the combined sets and for all 30 data sets, whereby we allowed different hyperparameter values for each experiment, to achieve the best possible results at the dataset level. The algorithms are sorted in descending order by their total average performance. To show the importance of the filters, we have evaluated the performance of the kNN graph with different values for k without any filters (CF in the table). *FC1*, *DBSCAN*, *FC2*, and *MkNN*

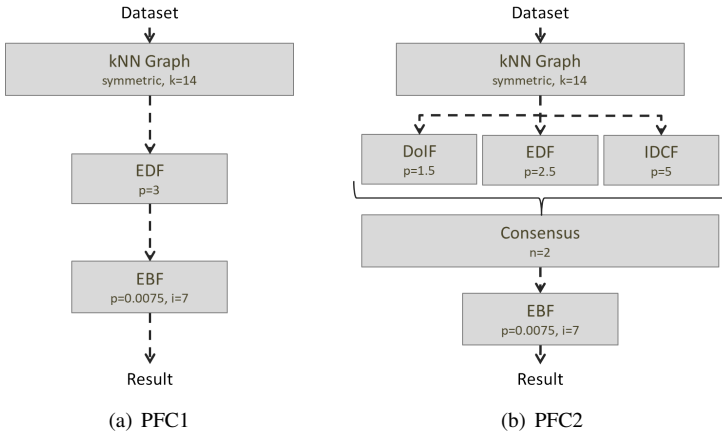


Fig. 4: Structure of predefined filter cascades 1 (PFC1) and 2 (PFC2).

Tab. 3: AVG F1-Score of Cluster Flow compared to the baseline. Left: Performance of all methods with varying parameters. Right: Performance of the best baseline algorithms, PFC1 and PFC2 with constant parameters.

Changing Parameters					Constant Parameters	
Algorithm	Set1	Set2	Set3	AVG total	Algorithm	AVG total
FC1	0.9972	0.9985	0.9754	0.9902	PFC1	0.995
DBSCAN	0.9967	0.9999	0.9716	0.9894	DBSCAN	0.828
FC2	0.9967	0.9938	0.9599	0.9834	PFC2	0.931
MkNN	0.9494	0.9990	0.9029	0.9504	MkNN	0.904
CF (no filters)	0.9224	0.9993	0.7979	0.9065		
CHAMELEON	0.8793	0.8377	0.8401	0.8519		
Spectral	0.9605	0.6857	0.8153	0.8205		
k-means	0.9360	0.6505	0.7407	0.7757		
Rock	0.6935	0.6124	0.7326	0.6795		

based clustering achieved the best results. On *set2*, MkNN performed a little bit better than the kNN clustering approaches. On the other two sets, FC1 and FC2 outperformed the MkNN clustering significantly. In total, FC1 achieved the best results with an average F1-score of 0.990, while DBSCAN came second with 0.989. However, FC1, FC2, and DBSCAN achieved very similar results on all sets apart from small fluctuations. DBSCAN is known to perform well on many of the selected sets. The goal here was to reveal that Cluster Flow consistently delivers better or equally strong results, even on data sets with distributions predestined for DBSCAN or other competitors. However, we also want to explicitly point out situations where our approach significantly outperforms DBSCAN, i.e., identifying clusters with varying densities. Therefore regard Figure 5: (a) shows the best

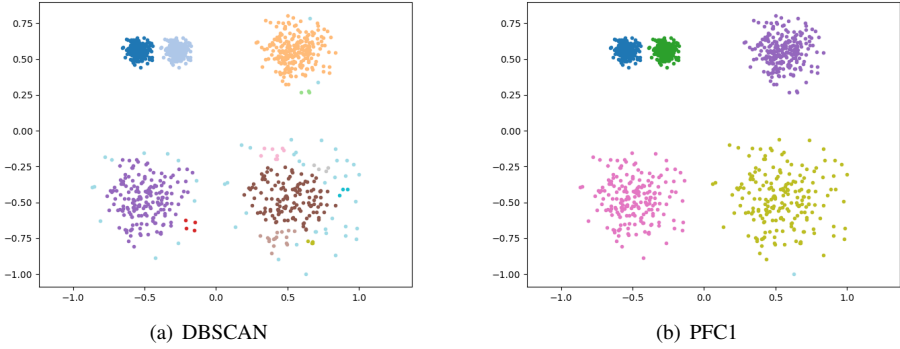


Fig. 5: Qualitative example where PFC1 outperforms DBSCAN as it is able to detect clusters of varying densities.

clustering result of DBSCAN after testing different parameter settings and (b) shows the clustering result of the fully automatically PFC1 on a self created dataset⁸ with varying densities.

5.5 Performance while maintaining constant parameters

In the previous analysis, we explicitly determined the optimal parameters for each algorithm and data set individually to achieve the best possible result. However, choosing the right parameters is a laborious and time-consuming task, especially for laymen, since the optimal hyperparameters can vary significantly from dataset to dataset. Thus, we evaluated the performance when using the same parameter settings for all 30 low dimensional data sets. As baseline we used DBSCAN ($\epsilon = 0.08$, $MinPoints = 3$) and MkNN ($k = 10$), as these gave the best results in the upper analysis. The right side of Table 3 summarizes the achieved results of each algorithm constraint to not changing parameters across all 30 benchmark datasets. Here, PFC1 and PFC2 outperformed the other algorithms. These results show the potential of predefined filter cascades in general and that PFC1 and PFC2 are well-suited to obtain a useful clustering without adjusting the parameters, especially without knowing the type or the distribution of the data in advance. Most algorithms only work for specific shapes and distributions of clusters but then fail for other cluster forms. In the real world, however, data distribution is not known in advance, so it is of great importance to offer clustering algorithms that can achieve consistently good results regardless of the distribution and shape of the clusters without having to tweak the hyperparameters. While PFC1 is more progressive in that it deletes all edges considered unwanted, PFC2 offers a more

⁸ Dataset *different_density* on <https://github.com/wahlflo/Datasets>

conservative approach, where a certain percentage of filters must share a consensus to force the deletion of edges.

We also performed experiments on the previously described high-dimensional datasets dim032, dim064, etc., which consist of well-separated, randomly sampled Gaussian clusters. In total, the F1-scores of PFC1 (**0.96, 0.93, 0.95, 0.97, 0.97**) were slightly better or equal to the scores of PF2 (0.95, **0.93**, 0.94, 0.94, 0.93). In general, both showed consistently good results.

6 Conclusion

We developed Cluster Flow, a new advanced concept to cluster data based on kNN graphs. Our approach's key components are modularity, which is also the key for offering intermediate interaction stages, explainability, and simultaneously identifying various cluster shapes. Experiments on more than 30 benchmark datasets show that the proposed technique consistently achieves superior results when used interactively, i.e., varying parameters for different datasets. On top of that, even not seen in an interactive context, the predefined filter cascades PFC1 and PFC2 can be used as entirely autonomous clustering algorithms that work fully automatically and achieved remarkable results over various experiments. Non-convex clusters are found as well as clusters of varying density. The easy to understand concept allows researchers from all areas with no previous knowledge in clustering to explore, cluster, and understand the data in depth. Hence, we have shown an efficient clustering concept that can successfully find diverse clusters and is highly understandable. As the focus of this paper is developing a concept of how to compose easy steps so that laymen can understand what their clustering and results mean, we leave a user study for an even more beautiful visualization and potentially better usability for future work. Additionally, in future work, one could integrate other data types than numerical data and investigate further filter and combination methods. Another goal is to generate branches within the interactive clustering workflow, i.e., to work with several independent intermediate states in parallel or use a change history. To further support the user in the decision process metadata of the nodes or other interesting properties could be displayed. Of course, current acceleration methods like accelerating the kNN graph computation [CLR20] could be integrated, too. For high dimensional data, kNN could be computed according to the subspace importance [Ba04].

Acknowledgments

This work has been funded by the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IS18036A. The authors of this work take full responsibilities for its content.

Bibliography

- [Ba04] Baumgartner, Christian; Plant, Claudia; Railing, K; Kriegel, H-P; Kroger, Peer: Subspace selection for clustering high-dimensional data. In: Fourth IEEE International Conference on Data Mining (ICDM'04). IEEE, pp. 11–18, 2004.
- [Ba20] Bae, Juhee; Helldin, Tove; Riveiro, Maria; Nowaczyk, Sławomir; Bouguelia, Mohamed-Rafik; Falkman, Göran: Interactive Clustering: A Comprehensive Review. *ACM Computing Surveys (CSUR)*, 53(1):1–39, 2020.
- [BKS19] Beer, Anna; Kazempour, Daniyal; Seidl, Thomas: Rock - Let the points roam to their clusters themselves. In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. pp. 630–633, 2019.
- [Br97] Brito, MR; Chavez, EL; Quiroz, AJ; Yukich, JE: Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35(1):33–42, 1997.
- [CLR20] Chávez, Edgar; Ludueña, Verónica; Reyes, Nora: Heuristics for Computing k-Nearest Neighbors Graphs. In: *Computer Science–CACIC 2019: 25th Argentine Congress of Computer Science, CACIC 2019, Río Cuarto, Argentina, October 14–18, 2019, Revised Selected Papers 25*. Springer, pp. 234–249, 2020.
- [CY08] Chang, Hong; Yeung, Dit-Yan: Robust path-based spectral clustering. *Pattern Recognition*, 41(1):191–203, 2008.
- [Es96] Ester, Martin; Kriegel, Hans-Peter; Sander, Jörg; Xu, Xiaowei et al.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: *Kdd*. volume 96, pp. 226–231, 1996.
- [FJ05] Fred, Ana LN; Jain, Anil K: Combining multiple clusterings using evidence accumulation. *IEEE transactions on pattern analysis and machine intelligence*, 27(6):835–850, 2005.
- [FS18] Fränti, Pasi; Sieranoja, Sami: K-means properties on six clustering benchmark datasets. *Applied Intelligence*, 48(12):4743–4759, 2018.
- [FVH06] Fränti, Pasi; Virtajoki, Olli; Hautamaki, Ville: Fast agglomerative clustering using a k-nearest neighbor graph. *IEEE transactions on pattern analysis and machine intelligence*, 28(11):1875–1881, 2006.
- [GMT07] Gionis, Aristides; Mannila, Heikki; Tsaparas, Panayiotis: Clustering aggregation. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):4–es, 2007.
- [GN02] Girvan, Michelle; Newman, Mark EJ: Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.
- [HHL10] Hahmann, Martin; Habich, Dirk; Lehner, Wolfgang: Visual decision support for ensemble clustering. In: *International Conference on Scientific and Statistical Database Management*. Springer, pp. 279–287, 2010.
- [HKF04] Hautamaki, Ville; Karkkainen, Ismo; Fränti, Pasi: Outlier detection using k-nearest neighbour graph. In: *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004*. volume 3. IEEE, pp. 430–433, 2004.

- [JL05] Jain, Anil K; Law, Martin HC: Data clustering: A user's dilemma. In: International conference on pattern recognition and machine intelligence. Springer, pp. 1–10, 2005.
- [KHK99] Karypis, George; Han, Eui-Hong; Kumar, Vipin: Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.
- [Li15] Liu, Hongfu; Liu, Tongliang; Wu, Junjie; Tao, Dacheng; Fu, Yun: Spectral ensemble clustering. In: Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining. pp. 715–724, 2015.
- [LI82] Lloyd, Stuart: Least squares quantization in PCM. *IEEE transactions on information theory*, 28(2):129–137, 1982.
- [MHVL07] Maier, Markus; Hein, Matthias; Von Luxburg, Ulrike: Cluster identification in nearest-neighbor graphs. In: International Conference on Algorithmic Learning Theory. Springer, pp. 196–210, 2007.
- [MLH09] Maier, Markus; Luxburg, Ulrike V; Hein, Matthias: Influence of graph construction on graph-based clustering measures. In: Advances in neural information processing systems. pp. 1025–1032, 2009.
- [Pe01] Pearson, Karl: LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.
- [SB14] Sardana, Divya; Bhatnagar, Raj: Graph clustering using mutual K-nearest neighbors. In: International Conference on Active Media Technology. Springer, pp. 35–48, 2014.
- [SG02] Strehl, Alexander; Ghosh, Joydeep: Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *Journal of machine learning research*, 3(Dec):583–617, 2002.
- [SM00] Shi, Jianbo; Malik, Jitendra: Normalized cuts and image segmentation. *Departmental Papers (CIS)*, p. 107, 2000.
- [SRS00] Sridhar, Ramaswamy; Rastogi, Rajeev; Shim, Kyuseok: Efficient algorithms for mining outliers from large data sets. In: International Conference on Management of Data: Proceedings of the 2000 ACM SIGMOD international conference on Management of data: Dallas, Texas, United States. volume 15, pp. 427–438, 2000.
- [VL07] Von Luxburg, Ulrike: A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.
- [VPRS11] Vega-Pons, Sandro; Ruiz-Shulcloper, José: A survey of clustering ensemble algorithms. *International Journal of Pattern Recognition and Artificial Intelligence*, 25(03):337–372, 2011.
- [VRB02] Veenman, Cor J.; Reinders, Marcel J. T.; Backer, Eric: A maximum variance cluster algorithm. *IEEE Transactions on pattern analysis and machine intelligence*, 24(9):1273–1280, 2002.
- [Wu13] Wu, Junjie; Liu, Hongfu; Xiong, Hui; Cao, Jie: A theoretic framework of k-means-based consensus clustering. In: Twenty-Third International Joint Conference on Artificial Intelligence. 2013.
- [Za71] Zahn, Charles T: Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on computers*, 100(1):68–86, 1971.

When Bears get Machine Support: Applying Machine Learning Models to Scalable DataFrames with Grizzly

Steffen Kläbe,¹ Stefan Hagedorn²



Abstract: The popular Python Pandas framework provides an easy-to-use DataFrame API that enables a broad range of users to analyze their data. However, Pandas faces severe scalability issues in terms of runtime and memory consumption, limiting the usability of the framework. In this paper we present Grizzly, a replacement for Python Pandas. Instead of bringing data to the operators like Pandas, Grizzly ships program complexity to database systems by transpiling the DataFrame API to SQL code. Additionally, Grizzly offers user-friendly support for combining different data sources, user-defined functions, and applying Machine Learning models directly inside the database system. Our evaluation shows that Grizzly significantly outperforms Pandas as well as state-of-the-art frameworks for distributed Python processing in several use cases.

1 Introduction

Python has become one of the most widely used programming language for Data Science and Machine Learning. According to the TIOBE index the languages popularity has steadily grown and was awarded *language of the year* in 2007, 2010, and 2018³. The popularity obviously comes from its easy-to-learn syntax which allows rapid prototyping and fast time-to-insight in data analytics.

Python's success is also founded in the vast amount of libraries that help developers in their tasks. Nowadays, the most popular framework for loading, processing, and analyzing data is the Pandas library. Pandas had a huge success as it has connectors to read data in different file formats and represents it in a unified DataFrame abstraction. The DataFrame implementation keeps data in memory and comes with a variety of operators to filter, transform, join the DataFrames or executing different kinds of analytical operations on the data. However, the in-memory processing of Pandas comes with serious limitations and drawbacks:

- Data sizes are limited to the main memory capacity of the client machine, as there is no way of automatic disk-spilling and buffer management as found in almost every database systems.

¹ Technische Universität Ilmenau, Germany, steffen.klaebe@tu-ilmenau.de

² Technische Universität Ilmenau, Germany, stefan.hagedorn@tu-ilmenau.de

³ <https://www.tiobe.com/tiobe-index/python/>, October 2020

- Even if the data to process resides in a database system on a powerful server, Pandas will load all data onto the data scientist's computer for processing. This is not only time consuming, but one can also assume that a companies sales table will quickly become larger than the memory of the data scientist's work station.
- Operations on a Pandas DataFrame often create copies of the DataFrame instead of performing the operation in place, occupying additional precious and limited memory.

In order to solve the memory problems, many users try to implement their own buffer manager and data partitioning strategies to only load parts of the original input file. However, we believe that scientists trying to find answers in the data should not be bothered with data management and optimization tasks, but this should rather be addressed by the storage and processing system.

Besides data analytics, Machine Learning models have become more and more popular during recent years. There are numerous frameworks for Python to create, train, and apply artificial neural network models on some input data. Often, these Machine Learning frameworks directly support Pandas DataFrames as input data. However, the programming effort to apply these models to data situated in arbitrary sources is high and not all users trained the models themselves, but want to use existing pre-trained models in their applications. This use case is therefore also of high importance in the field of data analytics, but not yet integrated into Pandas in an easy-to-use way.

Contribution In this paper we present our Grizzly⁴ framework, which provides a DataFrame API similar to Python Pandas, but instead of shipping the data to the program, the program is shipped to where the data resides.

In [Hag20] we sketched our initial idea of the Grizzly framework, a transpiler to generate SQL queries from a Pandas-like API. We argue that for many scenarios data is already stored in a (relational) database and used for different applications. Therefore, analysts using this data should neither be bothered with learning SQL to access this data nor with implementing buffer management strategies to be able to process this data with Pandas in Python. In this paper we present an extension to the initial overview in [HK21] by providing API extensions and make the following contributions:

- We present a framework that provides a DataFrame API similar to Pandas and transpiles the operations into SQL queries, moving program complexity to the optimized environment of a DBMS.
- The framework is capable of processing external files directly in the database by automatically generating code to use DBMS specific external data source providers. This especially enables the user to join files with the existing data in the database directly in the DBMS.

⁴ Available on GitHub: <https://github.com/dbis-ilm/grizzly>

- User-defined functions (UDFs) are also shipped to the DBMS by exploiting the support of the Python language for stored procedures of different database systems. By automatically generating the UDF code to apply the models to the data, Grizzly enables users to apply already trained Machine Learning models, e.g., for classification or text analysis to the data inside the database in a scalable way.

The remainder of the paper is organized as follows: We discuss related work and compare existing systems with a Pandas-like `DataFrame` API in Section 2. In Section 3 we present the architecture of our Grizzly framework as well as the transpilation of `DataFrame` operations to SQL code. Afterwards the important features of Grizzly are explained in detail, namely the external data source support in Section 4, the UDF support in Section 5 and the model join feature in Section 6. We evaluate the performance impact and scalability of Grizzly in Section 7 before concluding in Section 8.

2 Related work

There have been several systems proposed to translate user programs into SQL. The RIOT project [ZHY09] proposed the RIOT-DB to execute R programs I/O efficiently using a relational database. RIOT can be loaded into an R program as a package and provides new data types to be used, such as vectors, matrices, and arrays. Internally objects of these types are represented as views in the underlying relational database system. This way, operations on such objects are operations on views which the database system eventually optimizes and executes. Another project to perform Python operations as in-database analytics is AIDA [DDK18], but focuses mainly on linear algebra with NumPy as an extension besides relational algebra. The AIDA client API connects to the AIDA server process running in the embedded Python interpreter inside the DBMS (MonetDB) to send the program and retrieve the results. AIDA uses its `TabularData` abstraction for data representation which also serves to encapsulate the Remote Method Invocation of the client-server communication.

Several projects have been proposed to overcome the scalability and performance issues in the Pandas framework. These projects can be categorized by their basic approaches of optimizing the Python execution or transpiling the Pandas programs into other languages. Modin [Pet+20] is the state-of-the-art system for the Python optimization approach. By offering the same API as Pandas, it can be used as a drop-in-replacement. In order to accelerate the Python execution, it transparently partitions the `DataFrames` and compiles queries to be executed on Ray [Mor+18] or Dask⁵, two execution engines for parallel and distributed execution of Python programs. Additionally, Modin supports memory-spillover, so (intermediate) `DataFrames` may exceed main memory limits and are spilled to persistent memory. This solves the memory limitation problem of Pandas. However, Modin also uses eager execution like Pandas and still requires the client machine to consist of powerful hardware, since data from within a database system is fetched onto the client, too.

⁵ <https://www.dask.org/>

In the field of systems that use the transpiling approach, Koalas⁶ brings the Pandas API to the distributed spark environment using PySpark. It uses lazy evaluation and relies on Pandas UDFs for transpiling. The creators of the Pandas framework also tackle the problem of the eager client side execution in IBIS⁷. IBIS collects operations and converts them into a (sequence of) SQL queries. Additionally, IBIS can connect to several (remote) sources and is able to run UDFs for Impala or Google BigQuery as a backend. Though, tables from two different sources, such as different databases, cannot be joined within an IBIS program. With a slightly modified API, AFrame [SC19] transpiles Pandas code into SQL++ queries to be executed in AsterixDB. In contrast, in Grizzly we produce standard SQL which can be executed by any SQL engine and use templates provided in a configuration file to account for vendor-specific dialects.

An approach to integrate Machine Learning into columnar database systems was proposed in [Raa+18]. The approach uses handcrafted Python UDFs to train the model inside the database, store the model as a DBMS-specific internal serialized object and apply the model by deserializing it again. In comparison, Grizzly supports pre-trained, portable model formats, automatically generates code to apply the models and also introduces a caching approach to cache the model, which reduces the loading (or deserialization) overhead and is therefore of major importance for deep and complex neural networks.

The main features of the presented systems are compared to Grizzly in Tab. 1. Grizzly also uses the approach of transpiling Python code to SQL, making it independent from the actual backend system and therefore being more generic than Koalas or AFrame. Similar to the proposed systems, Grizzly provides an API similar to the Pandas DataFrame API with the goal to abstract from the underlying execution engine. However, Grizzly extends this API with two main features that clearly separates it from the other systems. First, it provides in-DBMS support for external files. This enables server-side joins of different data sources, e.g. database tables and flat files. As a consequence, performance increases significantly for these use cases compared to the client-side join of the sources that would be necessary in the other systems. Additionally, the result of the server-side join remains in the database, enabling subsequent operations to be also executed in the DBMS instead of the client machine. Second, Grizzly offers an easy-to-use API for applying Machine Learning models directly in the DBMS. Compared to the other systems where this feature could be simulated by handcrafting UDF code to apply the model on the client side, Grizzly exploits the UDF feature of DBMS and automatically generates code to cache the loaded model in memory and apply the pre-trained models directly on the server side. Furthermore, applying the model requires several (Python) functions, which can only be realized non-optimally using handcrafted UDFs. Again, as results remain in the DBMS, subsequent operations can be executed efficiently by the DBMS before returning the result to the client. Besides the performance aspect, this feature makes it significantly easier to apply Machine Learning models to the data compared to handcrafted UDFs.

⁶ <https://www.github.com/databricks/koalas>

⁷ <http://ibis-project.org/>

	Modin	Koalas	IBIS	AFrame	Grizzly
Approach	Python optimization	Transpiling	Transpiling	Transpiling	Transpiling
Backends	Ray, Dask	Spark	Arbitrary DBMS with SQL-API	AsterixDB	Arbitrary DBMS with SQL-API
Query Evaluation	Eager	Lazy	Lazy	Lazy	Lazy
UDF Support	On local Pandas DataFrames	Over Pandas UDFs	In-DBMS execution for Impala and BigQuery	In-DBMS execution for AsterixDB	In-DBMS execution for Postgres and Vector
Ext. File Support	Read to DataFrame	Read to DataFrame	Read to DataFrame	Read to DataFrame	In-DBMS support using ext. tables/foreign data wrappers
ML Model Support	Handcrafted UDFs	Handcrafted UDFs	Handcrafted UDFs	API for Scikit models	API for ONNX, Tensorflow, PyTorch

Tab. 1: Comparison of available systems with Pandas-like API.

3 Architecture

There are two major paradigms of data processing: data shipping and query shipping [Kos00]. While in the data shipping paradigm, as found in Pandas, the possibly large amount of data is transferred from the storage node to the processing node, in query shipping the query/program is transferred to where the data resides. The latter is found in DBMSs, but also in Big Data frameworks such as Apache Spark and Hadoop.

In this section we discuss the architecture of our Grizzly framework which is designed to maintain the ease-of-use of the data shipping paradigm in combination with the scalability of the query shipping approach. Grizzly is available as Open Source and in its core it consists of a DataFrame implementation and a Python-to-SQL transpiler. It is intended to solve the scalability issues of Pandas by transforming a sequence of operations on DataFrames into a SQL query that is executed by a DBMS. However, we would like to emphasize that the code generation and execution is realized using a plug-in design so that code generator for other languages than SQL or execution engines other than relational DBMSs (e.g., Spark or NoSQL systems) can be implemented and used. In the following, we show how SQL code generation is realized using a mapping between DataFrames and relational algebra.

Figure 1 shows the general (internal) workflow of Grizzly. As in Pandas, the core data structure is a DataFrame that encapsulates operations to compute result data. However, in Grizzly a DataFrame is only a hull and it does not contain the actual data. Rather, the operations on a DataFrame only create specific instances of DataFrames, such as

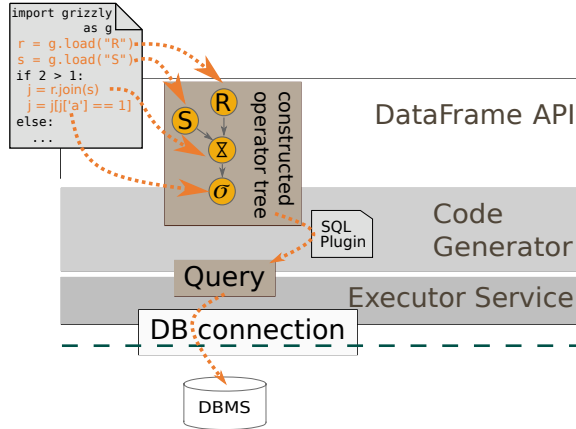


Fig. 1: Overview of Grizzly's architecture.

ProjectionDataFrame or FilterDataFrame, to track the operations. A DataFrame instance stores all necessary information required for its operation as well as the reference to the DataFrame instance(s) from which it was created. This lineage graph basically represents the operator tree as found in relational algebra. The leaves of this operator tree are the DataFrames that represent a table (or view) or some external file. Inner nodes represent transformation operations, such as projections, filters, groupings, or joins, and hence, their results are DataFrames again. The actual computation of the query result is triggered via actions, whose results are directly needed in the client program, e.g., aggregation functions which are not called in the context of `group by` clause. To view the result of queries that do not use aggregation, special actions such as `print` or `show` are available to manually trigger the computation.

Building the lineage graph of DataFrame modifications, i.e., the operator tree, follows the design goal of lazy evaluation behavior as it is also found in the RDDs in Apache Spark [Zah+12]. When an action is encountered in a program, the operator tree is traversed, starting from the DataFrame on which the action was called. While traversing the tree, for every encountered operation its corresponding SQL expression is constructed as a string and filled in a SQL template. For this, we apply a mapping of Pandas operations to SQL statements. This mapping is shown in Table 2. Based on the operator tree, the SQL query can be constructed in two ways:

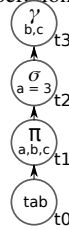
1. generate nested sub-queries for every operation on a DataFrame, or
2. incrementally extend a single query for every operation found in the Python program.

In Grizzly we implement variant (1), because variant (2) has the drawback to decide whether the SQL expression of an operation can be merged into the current query or a sub-query has to be created. Though, the SQL parser and optimizer in the DBMSs have been implemented and optimized to recognize such cases. As an example, Figure 2 shows how a Python script

	Python Pandas	SQL
Projection	<code>df['A']</code> <code>df[['A', 'B']]</code>	<code>SELECT a FROM ...</code> <code>SELECT a,b FROM ...</code>
Selection	<code>df[df['A'] == x]</code>	<code>SELECT * FROM ...WHERE a = x</code>
Join	<code>pandas.merge(df1, df2,</code> <code>left_on='x', right_on='y',</code> <code>how='inner outer right left')</code>	<code>SELECT * FROM df1</code> <code>inner outer right left join df</code> <code>ON df1.x = df2.y</code>
Grouping	<code>df.groupby(['A', 'B'])</code>	<code>SELECT * FROM ...GROUP BY a,b</code>
Sorting	<code>df.sort_values(by=['A', 'B'])</code>	<code>SELECT * FROM ...ORDER BY a,b</code>
Union	<code>df1.append(df2)</code>	<code>SELECT * FROM df1</code> <code>UNION ALL SELECT * FROM df2</code>
Intersection	<code>pandas.merge(df1, df2,</code> <code>how='inner')</code>	<code>SELECT * FROM df1</code> <code>INTERSECTION SELECT * FROM df2</code>
Aggregation	<code>df['A'].min()</code> <code>max() mean() count() sum()</code> <code>df['A'].value_counts()</code>	<code>SELECT min(a) FROM ...</code> <code>max(a) avg(a) count(a) sum(a)</code> <code>SELECT a, count(a) FROM ...</code> <code>GROUP BY a</code>
Add column	<code>df['new'] = df['a'] + df['b']</code>	<code>SELECT a + b AS new FROM ...</code>

Tab. 2: Basic Pandas DataFrame operations and their corresponding SQL statements.

```
# load table (t0)
df = grizzly.read_table("tab")
# projection to a,b,c (t1)
df = df[['a', 'b', 'c']]
# selection (t2)
df = df[df.a == 3]
# group by b,c (t3)
df = df.groupby(['b', 'c'])
```



```
SELECT t3.b, t3.c FROM (
  SELECT * FROM (
    SELECT t1.a, t1.b, t1.c FROM (
      SELECT * FROM tab t0
    ) t1
  ) t2 WHERE t2.a = 3
) t3 GROUP BY t3.b, t3.c
```

(a) Source Python code.

(b) Operator tree.

(c) Produced SQL query.

Fig. 2: Steps for transpiling Python code to a SQL query: The operations on DataFrames (a) are collected in an intermediate operator tree (b) which is traversed to produce a nested SQL query (c).

is transformed into a SQL query. Although the nested query imposes some overhead to the optimizer for unnesting in the DBMS and bears the risk that it fails to produce an optimal query, we believe they are very powerful and mostly well tested, so that it is not worth it to re-implement such behavior in Grizzly. The generated query is sent to a DBMS using a user-defined connection object, as it is typically used in Python and specified by PEP 249⁸. Grizzly produces standard SQL with vendor-specific statements to create functions or access external data as we will discuss below. The vendor-specific statements are taken from templates defined in a configuration file. By providing templates for the respective functions one can easily add support for arbitrary DBMSs. We currently support Actian Vector and PostgreSQL.

Besides the plain SQL queries, Grizzly needs to produce additional statements in order to

⁸ <https://www.python.org/dev/peps/pep-0249/>

set up user-defined functions and connectors to external data sources. If the Python code uses, e.g., UDFs, this function must be created in the database system before it can be used in the query. Thus, Grizzly produces a list of so-called pre-queries. A pre-query to create a UDF is the `CREATE FUNCTION` statement including the corresponding function name, input and output parameters as well as the function body of course. For an external data source, the pre-query creates the necessary DBMS-specific connection to the data source, as described in the next section.

One design goal of the Grizzly framework is to serve as a drop-in replacement for Pandas in the future. Being under active development, we did not yet reach a state of full API compatibility. For operations that are not supported yet or can not be expressed in SQL, one might fallback to either Pandas operations by triggering the execution inside the DBMS and proceed with the intermediate result in Pandas, or exploit the Python UDF feature of modern DBMS to execute operations as described in Section 5.

4 Support for External Data Sources

In typical data analytics tasks data may be read from various formats. On the one hand, (relational) database systems are used to store and archive large data sets like company inventory data or sensor data in IoT applications. On the other hand, data may be created by hand, exported from operational systems or shared as text files like CSV or JSON. For these files it is not always necessary, intended or beneficial to import them into a database system first, as they might be only for temporary usage or need to be analyzed before loading them into the database. As a consequence, there is a gap between tables stored in a database system and plain files in the filesystem, and both sources need to be combined. In Pandas, one would need to read the data from the database as well as the text files and combine them in main memory. Since it is our goal to shift the complete processing into the DBMS, the files need to be transferred and imported into the DBMS transparently. In our framework, we achieve this by using the ability of many modern DBMS to define a table over an external file as defined in the SQL/MED standard from 2003 [Mel+02].

As an example, PostgreSQL offers *foreign data wrappers* (FDW) to access external sources such as files, but also other database systems. A PostgreSQL distribution includes FDWs for, e.g., CSV files, files in HDFS as well as other relational and non-relational DBMSs. Own FDWs for other sources can easily be installed as extensions. Internally, the planner uses the access costs to decide for the best access path, if such information is provided by the FDW.

Besides PostgreSQL, Actian Vector offers the external table feature, which is realized using the Spark-Vector-Connector⁹. Here Vector handles external tables as meta data in the catalog with a reference to a file path in the local filesystem or HDFS. Whenever a query accesses an external table, Vector exploits the capabilities of Apache Spark to read data efficiently in parallel, leading to fast scans of external tables.

⁹ <https://github.com/ActianCorp/spark-vector>

In Grizzly, we offer easy-to-use operations to access external data sources. These operations return a `DataFrame` object and are the leaves of the operator lineage graph described in Section 3, similar to ordinary database tables. Here a user has to specify the data types of the data in the text files as well as the file path. During SQL code generation, a pre-query is automatically generated that creates the external table/foreign data wrapper for each of these leaves. As the syntax of these queries might be vendor-specific, we maintain templates to create an external data source in the configuration file. The pre-queries are then appended to the pre-query list described in Section 3, so they are ensured to be executed before the actual analytical query is run. In the actual query, these tables are then referenced using their temporary names.

An important point to highlight here is that the database server must be able to access the referenced file. We argue that with network file systems mounts, NAS devices or cloud file systems this is often the case. Even actively copying the file to the server is not a problem since such data files are rather small, compared to the amount of data stored in the database.

5 Support for Python UDFs

Another important part of data analytics is data manipulation using user-defined functions. In pandas, users can create custom functions and apply them to `DataFrames` using the `map` function. Such functions typically perform more or less complex computations to transform values or combine values of different columns. These UDFs are a major challenge when transpiling Pandas code to SQL, as their definitions must be read and transferred to the DBMS. This requires that the Python program containing the Pandas operations can somehow access the function's source code definition. In Python, this can be done via reflection tools¹⁰. Most DBMS support stored procedures and some of them, e.g., PostgreSQL and Actian Vector, also allow to define them using Python (language PL/Python). This way, functions defined in Python are processed by Grizzly, transferred to the DBMS and dynamically created as a (temporary) function. Note that most systems only offer scalar UDFs at the moment, which produce a single output tuple from a single input tuple. Consequently, Grizzly only supports this class of functions and does not offer any support for table UDFs, which produce an output tuple for an arbitrary number of inputs.

The actual realization of the UDF support is hereby different for different DBMS and shows some limitations that needs to be considered. First, Python UDFs are only available as a beta version in PostgreSQL and Actian Vector. The main reason for this is that there are severe security concerns about using the feature, as especially sandboxing a Python process is difficult. As a consequence, users must have superuser access rights for the database or demand access to the feature from the administrator in order to use the Python UDF feature. While this might be a problem in production systems, we argue that this should not be an issue in the scientific use cases where Python Pandas is usually used for data analytics.

¹⁰ Using the `inspect` module: <https://docs.python.org/3/library/inspect.html>

Second, the actual architecture of running Python code in the database differs in the systems. While some systems start Python processes per-query, other systems keep processes alive over the system uptime. The per-query approach has the advantage that it offers isolation in the Python code between queries, which is important for ACID-compliance. As a drawback, the isolation makes it impossible to cache user-specific data structures in order to use it in several queries, which is of major importance when designing the model join feature in Section 6. On the contrary, keeping the Python processes alive allows to cache such a user context and use it in several queries. However, this approach violates isolation, so UDF code has to be written carefully to avoid side effects that might impact other queries.

Although the DBMS supports Python as a language for user defined code, SQL is a strictly typed language whereas Python is not. In order to get type information from the user's Python function, we make use of *type hints*, introduced in Python 3.5. A Python function using type hints looks like this:

```
def repeat(n: int, s: str) -> str:
    r = n*s # repeat s n times
    return r
```

Such UDFs can be used, e.g., to transform, or in this example case combine, columns using the map method of a `DataFrame`:

```
# apply repeat on every tuple using columns name, num as input
df['repeated'] = df[['num', 'name']].map(repeat)
```

Using the type hints and a mapping between Python and SQL types, Grizzly's code generator can produce a pre-query to create the function on the server. For PostgreSQL, the generated code is the following:

```
CREATE OR REPLACE FUNCTION repeat(n int, s varchar(1024))
RETURNS varchar(1024)
LANGUAGE plpython3u
AS 'r = n*s # repeat s n times
return r'
```

Currently, we statically map variable-sized Python types to reasonable big SQL types, which is a real limitation and should be improved in the future. The command to create the function in the system is vendor-specific and therefore taken from the config file for the selected DBMS. We then extract the name, input parameters, source code and return type using Python's `inspect` module and use the values to fill the template. The function body is also copied into the template. Similar to external data sources in Section 4, the generated code is appended to the pre-query list and executed before the actual query. The map operation is translated into a SQL projection creating a computed column in the actual query:

```
SELECT t0.*, repeat(t0.num, t0.name) as repeated
FROM ... t0
```

As explained above, the previous operation from which `df` was derived will appear in the FROM clause of this query.

6 Machine Learning Model Join

In Grizzly, we expand the API of Python Pandas with the functionality to apply different types of Machine Learning models to the data. In the following, we name this operation of applying a model to the data a “model join”. Instead of realizing this over a map-function in Pandas, which leads to a client-side execution of the model join and therefore faces the same scalability issues as Pandas, we exploit the recent upcome of user-defined functions in popular database management systems and realize the model join functionality using Python UDFs. As a consequence, we achieve a server-side execution of the model join directly in the database system, allowing automatic parallel and distributed computation.

Note that we talk about the usage of pre-trained models in this section, as database systems are not optimized for model training. However, applying the model directly in the database has the advantage that users can make use of the database functionality to efficiently perform further operations on the model outputs, e.g., grouping or filters. Additionally, users may use publicly available, pre-trained models for various use cases. For our discussions, we assume that necessary Python modules are installed and the model files are accessible from the server running the database system. In the following, we describe the main ideas behind the model join concept as well as details for the supported model types, their characteristics and their respective runtime environments, namely PyTorch, Tensorflow, and ONNX.

6.1 Model join concept

Performing a model join on an `DataFrame` triggers the generation of a pre-query as described in Section 3, which performs the creation of the respective database UDF. As the syntax for this operation is vendor-specific, the template is also taken from the configuration file. The generated code hereby has four major tasks:

1. Load the provided model.
2. Convert incoming tuples to the model input format.
3. Run the model.
4. Convert the model output back to an expected output format.

While steps 2-4 have to be performed for every incoming tuple, the key for an efficient model join realization is caching the loaded model in order to perform the expensive loading only if necessary. (Re-)Loading the model is necessary if it is not cached yet or if the model changed. These cases can be detected by maintaining the name and the timestamp of the model file. However, such a caching mechanism must be designed carefully under consideration of the different, vendor-specific Python UDF realizations discussed in Section 5.

We realize the caching mechanism by attaching the loaded model, the model file name and the model time stamp to a globally available object, e.g., an imported module in the UDF. The model is loaded only if the global object has no model attribute for the provided model

file yet or the model has changed, which is detected by comparing the cached timestamp with the filesystem timestamp. In order to avoid that accessing the filesystem to get the file modification timestamp is performed for each call of the UDF (and therefore for every tuple), we introduce a magic number into the UDF. The magic number is randomly generated for each query by Grizzly and cached in the same way as the model metadata. In the UDF code, the cached magic number is compared to the magic number passed and only if they differ, the modification timestamps are compared and the cached magic number is overwritten by the passed one. As a result, the timestamps are only compared once during a query, reducing the number of file system accesses to one instead of once-per-tuple. With this mechanism, we automatically support both Python UDF realizations discussed in Section 5, although the magic number and timestamp comparisons are not necessary in the per-query approach, as it is impossible here that the model is cached for the first tuple. We exploit the isolation violation of the second approach that keeps the Python process alive and carefully design the model join code to only produce the caching of the model and respective metadata as intended side effects.

6.2 Model types

Grizzly offers support for PyTorch¹¹, Tensorflow¹² and ONNX¹³ models. All three model formats have in common, that the user additionally needs to specify model-specific conversion functions for their usage in order to specify how the expected model input is produced and the model output should be interpreted. These functions are typically provided together with the model by creators. With A, B, C, D being lists of data types, the conversion functions have signatures $in_conv : A \rightarrow B$ and $out_conv : C \rightarrow D$, if the model converts inputs of type B into outputs of type C . With A and D being set as type hints, the overall UDF signature can be inferred as $A \rightarrow D$ as described in Section 5. With this, applying a model to data stored in a database can be done easily and might look like the example in Listing 1.

As the conversion functions are typically provided along with the model, users only need to write a few lines of code. It is thereby mandatory to specify the input parameter types of the `input_to_model` function as well as the output type of the `model_to_output` function. In this example, the resulting UDF would have signature `str -> str`. Running this example, Grizzly automatically generates the UDF code and triggers its creation in the DBMS before executing the actual query. The produced query along with the pre-query to setup the UDFs in PostgreSQL is shown in Listing 2.

The actual code for model application is generated from templates and varies for the different model types described in the following.

PyTorch The PyTorch library is based on the Torch library, originally written in Lua. PyTorch was presented by Facebook in 2016 and has gained popularity as it enabled

¹¹ <https://www.pytorch.org/>

¹² <https://www.tensorflow.org/>

¹³ <https://www.github.com/onnx/>

```
def input_to_model(a: str):
    ...

def model_to_output(a) -> str:
    ...

df = grizzly.read_table('tab') # load table
# apply model to every value in column 'col'
# using provided input and output conversion functions
# store model output in computed column 'classification'
df['classification'] = df['col'].apply_model("/path/to/model", input_to_model,
    ↪ model_to_output)
# group by e.g. predicted classes
df = df.groupby(['classification']).count()
df.show()
```

Listing 1: Python code of model join example

```
CREATE OR REPLACE FUNCTION apply_model_123(col varchar(1024))
RETURNS varchar(1024)
LANGUAGE plpython3u AS 'def input_to_model(a: str):
    ...

def model_to_output(a) -> str:
    ...

#apply model here
' parallel safe;

SELECT t2.classification, count(*) FROM (
  SELECT *, apply_model_123(t1.col) as classification FROM (
    SELECT * FROM tab t0
  ) t1
) t2 GROUP BY t2.classification
```

Listing 2: Generated SQL code of model join example

programs to utilize GPUs and integrate other famous Python libraries. In its core, PyTorch consists of various libraries for Machine Learning that have different functionality.

A trained model can be saved for later reuse. For saving, two options exist. The first option is to serialize the complete model to disk. This has the disadvantage that the model class definition must be available for the runtime when the model is loaded. In Grizzly, this would mean that users who want to use a pretrained model in their program also need the source code of the model class. The second option for storing a trained model is to store only the learned parameters. Although this option is more efficient during deserialization, the user code must explicitly create an instance of the model class. Thus, the source code of the model class must be available for the end user. Additionally, in order to instantiate the model

class, users need to provide initial parameters values to the model's constructor which may be unknown and hard to set for inexperienced users.

Besides the challenges for using PyTorch with foreign models, Grizzly allows to load PyTorch models where only the learned parameters have been stored (option 2 from above).

Tensorflow Tensorflow is a another famous framework for building, training and running Machine Learning models. Tensorflow models are directed, acyclic graphs (DAGs) that are statically defined. With placeholder variables attached to nodes of the model graph, inputs and outputs can be mapped to the graph nodes. A `tensorflow.Session` object is used as the main entrance point and allows to run the model after configuring.

During the training phase, arbitrary states of the model graph can be exported as a checkpoint, which is a serialized format of the graph and its properties. Grizzly supports these checkpoints as an model type and generates code to restore the model graph as well as the `tensorflow.Session` object. However, users have to know the names of placeholders defined in the model in order to map inputs and outputs to the respective model nodes. Additionally, users can specify a vocabulary file to translate inputs to an expected model input format if necessary. As these restrictions require in-depth knowledge about the model, Grizzly offers additional possibilities to automatically generate the conversion functions. Nevertheless, this harms the ease-of-use slightly.

Starting with version 2, Tensorflow offers different possibilities to exchange trained models with the introduction of the Tensorflow Hub¹⁴ library or support for the Keras¹⁵ framework. In the future, we aim at integrating support for these formats in Grizzly.

ONNX ONNX is a portable and self-contained format for model exchange, that is able to be executed with different runtime backends like Tensorflow, PyTorch or the `onnxruntime`¹⁶. The self-containment and the portability of models makes the ONNX format easy to use, which meets the design goals of Grizzly. In the generated model code, we rely on the `onnxruntime` as execution backend in order to be independent from Tensorflow or PyTorch. A broad collection of pre-trained models along with their conversion functions is available in the Model Zoo¹⁷.

7 Evaluation

In this Section, we compare our proposed Grizzly framework against Pandas version and Modin, the current state-of-the-art framework for distributed processing of Pandas scripts, as the other related systems presented in Section 2 do not offer all evaluated features. We present different experiments for data access as well as applying a machine learning model

¹⁴ <https://www.tensorflow.org/hub>

¹⁵ <https://www.keras.io/>

¹⁶ <https://www.github.com/microsoft/onnxruntime>

¹⁷ <https://www.github.com/onnx/models>

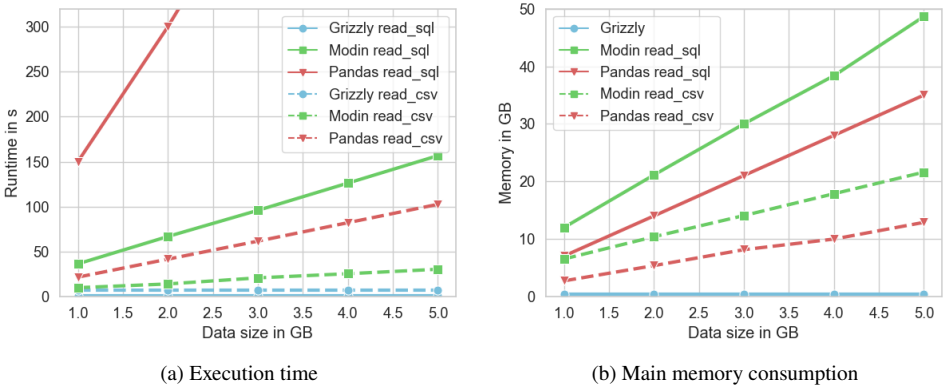


Fig. 3: CPU and RAM consumption for Pandas, Modin and our proposed Grizzly framework.

in a model join. Our experiments were run on a server consisting of a Intel(R) Xeon(R) CPU E5-2630 with 24 threads at 2.30 GHz and 128 GB RAM. This server runs Actian Vector 6.0 in a docker container, Python 3.6 and Pandas 1.1.1. Additionally we used Modin version 0.8 and experimentally configured it to the best of our knowledge, resulting in using Ray as the backend, 12 cores and out-of-core execution. For fairness, we ran the Pandas/Modin experiments on the same machine. As this reduces the transfer costs when reading tables from the database server, this assumption is always in favor of Pandas and Modin.

During our experiments, we discovered a bug in the parallel `read_sql` implementation of Modin, which produces wrong results for partitioned databases. The developers confirmed the issue and planned a fix for the next release. However, we used the parallel `read_sql` in order to not penalize Modin in the experiments, not considering the wrong results. This assumption is therefore also in favor of the Modin results.

7.1 Data access scalability

In this first experiment, we want to prove our initial consideration of Pandas' bad scalability with a minimal example use case. We used Actian Vector as the underlying database system and ran a query that scans data with varying size from a table or a csv file and performs a `min` operation on a column to reduce the result size. This way, we can compare the basic `read_sql` and `read_csv` operations of Pandas and Modin against Grizzly.

Figure 3 shows the execution time as well as the memory consumption of the evaluated query. For `sql` table access, Pandas shows an enormous runtime, linearly growing to 800 s for a data set size of 5 GB. Modin is significantly faster than Pandas and scales better. However, Grizzly is able to answer this query in a constant, sub-second runtime, as only the result has to be transferred to the client instead of the full dataset. Additionally, this query can be answered by querying small materialized aggregates [Moe98], which are used by default as an additional index structure in Vector. In comparison to Pandas/Modin, this

shows that with Grizzly queries can also benefit from index structures and other techniques to accelerate query processing used in database systems. For `csv` access, we can observe a similar behavior of Modin being faster and scaling better than Pandas due to its parallel `read_csv` implementation. Grizzly again shows a nearly constant runtime slightly higher than the `sql table` access but faster than Modin and Pandas. The main reason for this is that Grizzly uses the external table feature of Actian Vector, which is based on Apache Spark. As a consequence, the runtime is composed of the fixed Spark delays like startup or cleanup [WK15] and a variable runtime for reading the file in parallel. As data sizes are small here, the fixed Spark delays dominate the runtime.

Regarding memory consumption, Pandas again scales very poorly and memory consumption increases very fast, with the `read_sql_table` consuming more memory than the `read_csv` operation for the same data size. As a result, the memory consumption might exceed the available RAM of a client machine even for a small dataset size. In comparison, Modin consumes even more memory than Pandas for `read_sql` and `read_csv` respectively, potentially caused by the multiple worker threads. The memory consumption of Grizzly is mainly impacted by the result size, which is very small due to the choice of the query. However, this is only half of the truth, as Grizzly shifts the actual work to the DBMS which also consumes memory. Nevertheless, modern DBMS are designed for high scalability and are able to handle out-of-memory cases with buffer eviction strategies in the bufferpool or disk-spilling operators for database operators with a high memory consumption. Therefore, this is the ideal environment to run complex queries, as it is not limited to the available memory of the machine. Note that Modin also supports out-of-memory situations by disk-spilling. Another advantage of Grizzly is that the DBMS can run on a remote machine while the actual Grizzly script is executed from a client machine, which is then allowed to have an arbitrary hardware configuration while still being able to run complex analytics.

7.2 Combining data sources

An important task of data analysis is combining data from different data sources. We investigated a typical use case, namely joining flat files with existing database tables. We base our example on the popular TPC-H benchmark dataset [BNE14] on scale factor SF100, which is a typical sales database and is able to generate inventory data as well as update sets. We draw the following use case: The daily orders (generated TPC-H update set) are extracted from the productive system and provided as a flat file. Before loading them into the database system, a user might want to analyze the data directly by combining it with the inventory data inside the database. As an example query, we join the daily orders as a flat file with the customer table (1.5M tuples) from the database and determine the number of orders per customer market segment using an aggregation. The evaluated Python scripts are similar except the data access methods. While Pandas and Modin use (parallel) `read_sql` and `read_csv` for table and flat file access, Grizzly uses a `read_table` and a `read_external_table` call. This way, an external table is generated in Actian Vector, encapsulating the flat file access. Afterwards, the join as well as the aggregation are processed in the DBMS, and only the result is returned to the client.

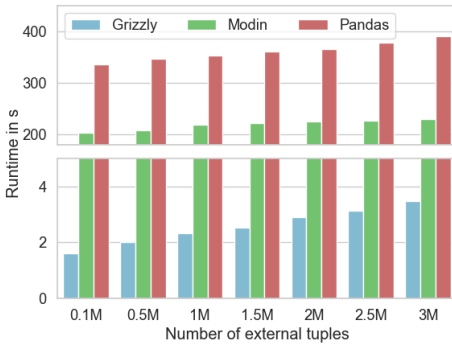


Fig. 4: Query runtime with database tables and external sources

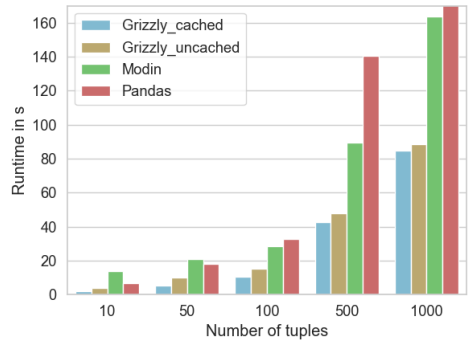


Fig. 5: Runtime for model join query.

For the experiment, we varied the number of tuples in the flat files. The runtime results in Figure 4 show that Grizzly achieves a significantly better runtime than Pandas and Modin. Additionally, it shows that Pandas and Modin suffer from a bad `read_sql` performance, as the runtime is already quite slow for small number of external tuples. Regarding scalability we can observe that runtime in Pandas grows faster with increasing number of external tuples than in Grizzly, caused by the fast processing of external tables in Actian Vector. Overall we can conclude that Grizzly significantly outperforms Python Pandas and Modin in this experiment and offers the possibility to process significantly larger datasets.

7.3 Model Join

There are various applications and use cases where machine learning models can be applied. As an example use case, we applied a sentiment analysis to a string column. We therefore used the IMDB dataset [Maa+11], which contains movie reviews, and applied the state-of-the-art RoBERTa model [Liu+19] with ONNX. The investigated query applies the model to the review column and groups on the output sentiment afterwards, counting positive and negative review sentiments. In Grizzly, we therefore use model join feature described in Section 6, while we handcrafted the function to apply the model for Modin and Pandas and invoked the function over the `map` function on the DataFrames after reading from the database.

Figure 5 shows the resulting runtimes for different number of review tuples. First, we can observe that Grizzly significantly outperforms Modin and Pandas in terms of runtime and scalability. For increasing data size Modin is also significantly faster and scales better than Pandas, while showing some overhead over Pandas for the very small data sets, as parallelism here introduces more overhead than benefit. While Modin achieves a speedup by splitting the DataFrames into partitions and applying the machine learning model in parallel, Grizzly achieves the performance improvement by applying the model directly in the database system. In Actian Vector, we used a parallel UDF feature and configured it for 12 parallel UDF workers, showing the best results for our setup in this experiment.

Additionally, with the Python implementation of Action Vector, which keeps the Python interpreters alive between queries, it is possible to reuse a cached model from a former query, leading to an additional performance gain of around 5 seconds.

7.4 Resume

In our evaluation, we proved the superiority of our proposed Grizzly framework over Pandas and Modin, the existing state-of-the-art distributed `DataFrame` framework. In different experiments for data access, the combination of different data sources and the application of pre-trained machine learning models to a dataset we showed that Grizzly has significantly better query performance as well as scalability, as most operations are executed directly in the database system. This is reinforced by the fact that query performance benefits from index structures of the DBMS, while these indexes do not have any impact when only reading data in a Pandas script and performing operations on the client. Furthermore, Grizzly enables complex data analysis tasks even on client machines without powerful hardware by pushing operations towards database servers. Consequently, queries are not limited to client memory, but are executed inside database systems that are designed to handle out-of-memory situations.

8 Conclusion

In this paper we presented Grizzly, a scalable and high-performant data analytics framework that offers a similar `DataFrame` API like the popular Pandas framework. As Pandas faces some severe scalability problems in terms of memory consumption and performance, Grizzly transpiles the easy-to-write Pandas code into SQL in order to push complexity to arbitrary database systems. This way, query execution is done in a scalable and highly optimized environment that is able to handle large datasets and complex queries. Additionally, by pushing queries towards remote database systems, complex analytics can be performed on client machines without extensive hardware requirements.

We extended the Pandas `DataFrame` API with several features in order to perform typical data analytics challenges in an efficient and easy-to-use way. First, Grizzly provides support for external data sources by exploiting the respective feature of several database systems and automatically generating code to create necessary tables or wrappers. This way, different sources like database tables or flat files can be combined and processed directly inside the DBMS instead of loading them into the client, improving performance and scalability. Second, Grizzly makes use of the recent upcome of Python user-defined functions in database systems in order to transparently push the execution of such functions on `DataFrames` into the DBMS. Third, applying pre-trained Machine Learning models to data is supported by Grizzly by automatically generating UDF code for Tensorflow, PyTorch or ONNX models. Pushing these operations to the database system not only increases performance and scalability, but also enables efficient processing of further operations on the model outputs, as they still remain in the database system. This allows a seamless integration of Machine Learning functionalities towards the vision of ML systems [Rat+19].

In our evaluation, we compared our proposed Grizzly framework to Pandas and Modin, the current state-of-the-art framework for distributed execution of Pandas-like DataFrames. In our experiments on data access scalability, combining different data sources, and applying Machine Learning models we proved that Grizzly significantly outperforms both systems while offering higher scalability and an easy-to-use API.

For the execution of UDFs and the application of Machine Learning models we rely on the Python UDF realization of database vendors. Most of them released Python UDFs only as a beta version until now as they faced security issues as well as performance issues. In the future, we monitor the development of this feature and aim at actively removing UDF execution performance as a choke point of query performance.

Additionally, we plan to investigate a different way of handling heterogeneous data sources in the future. Users may use hybrid warehouse approaches consisting of cloud database instances as well as on-premise instances to ensure data privacy and data security. This challenges a frontend framework like Grizzly to query multiple database instances and combine the results on the client side. In a conceptual view, we plan to extend Grizzly with an embedded query engine, e.g. DuckDB [RM19] or MonetDBLite [RM18], and a query optimizer and compare it against existing solutions like the Avalanche hybrid data warehouse¹⁸ or Polystores [Gad+16]. In the query graphs that Grizzly is based on, join operations between different database instances can then be seen as “pipeline breakers”, where data needs to be fetched from both join sides and combined locally. Using the query optimizer, computation effort needs to be pushed into the database instances as much as possible to reduce intermediate result sizes, transfer costs and the client-side processing costs.

Grizzly can be used in Jupyter Notebooks, where operations are typically performed incrementally. This incremental way of computation offers possibilities to further optimize the Grizzly workflow by, e.g., exploiting materialized views for intermediate results.

References

- [Moe98] Guido Moerkotte. “Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing”. In: *PVLDB*. Ed. by Ashish Gupta, Oded Shmueli, and Jennifer Widom. Morgan Kaufmann, 1998, pp. 476–487.
- [Kos00] Donald Kossmann. “The State of the Art in Distributed Query Processing”. In: *ACM Comput. Surv.* 32.4 (Dec. 2000), pp. 422–469. ISSN: 0360-0300.
- [Mel+02] Jim Melton et al. “SQL/MED - A Status Report”. In: *SIGMOD Rec.* 31.3 (2002), pp. 81–89.
- [ZHY09] Yi Zhang, Herodotos Herodotou, and Jun Yang. “RIOT: I/O efficient numerical computing without SQL”. In: *CIDR*. 2009.

¹⁸ <https://www.actian.com/analytic-database/avalanche/>

- [Maa+11] Andrew L. Maas et al. “Learning Word Vectors for Sentiment Analysis”. In: *Proc. of the 49th Annual Meeting of the Assoc. for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA, 2011, pp. 142–150.
- [Zah+12] Matei Zaharia et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *USENIX*. 2012.
- [BNE14] Peter A. Boncz, Thomas Neumann, and Orri Erling. “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”. en. In: *Performance Characterization and Benchmarking*. Springer, 2014, pp. 61–76.
- [WK15] K. Wang and M. M. H. Khan. “Performance Prediction for Apache Spark Platform”. In: *HPCC/CSS/ICSS*. 2015, pp. 166–173.
- [Gad+16] Vijay Gadepally et al. “The BigDAWG Polystore System and Architecture”. In: *HPEC* (Sept. 2016), pp. 1–6.
- [DDK18] Joseph Vinish D’Silva, Florestan D. De Moor, and Bettina Kemme. “AIDA - Abstraction for advanced in database analytics”. In: *VLDB* 11.11 (2018), pp. 1400–1413. issn: 21508097.
- [Mor+18] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI Applications”. en. In: *arXiv:1712.05889 [cs, stat]* (Sept. 2018). arXiv: 1712.05889.
- [RM18] Mark Raasveldt and Hannes Mühleisen. *MonetDBLite: An Embedded Analytical Database*. 2018. arXiv: 1805.08520 [cs.DB].
- [Raa+18] Mark Raasveldt et al. “Deep Integration of Machine Learning Into Column Stores”. In: *EDBT*. OpenProceedings.org, 2018, pp. 473–476.
- [Liu+19] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL].
- [RM19] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database”. en. In: *SIGMOD*. Amsterdam, Netherlands: ACM Press, 2019, pp. 1981–1984. isbn: 978-1-4503-5643-5.
- [Rat+19] A. Ratner et al. “SysML: The New Frontier of Machine Learning Systems”. In: *CoRR* abs/1904.03257 (2019). _eprint: 1904.03257.
- [SC19] Phanwadee Sinthong and Michael J. Carey. “AFrame: Extending DataFrames for Large-Scale Modern Data Analysis”. In: *Big Data*. Dec. 2019, pp. 359–371.
- [Hag20] Stefan Hagedorn. “When sweet and cute isn’t enough anymore: Solving scalability issues in Python Pandas with Grizzly”. In: *CIDR*. 2020.
- [Pet+20] Devin Petersohn et al. “Towards Scalable Dataframe Systems”. en. In: *arXiv:2001.00888 [cs]* (June 2020). arXiv: 2001.00888.
- [HK21] Stefan Hagedorn and Steffen Kläbe. “Putting Pandas in a Box”. In: *CIDR*. 2021.

Data Integration, Semantic Data Management, Streaming

Extended Affinity Propagation Clustering for Multi-source Entity Resolution

Stefan Lerm,¹ Alieh Saeedi² Erhard Rahm³

Abstract: Entity resolution is the data integration task of identifying matching entities (e.g. products, customers) in one or several data sources. Previous approaches for matching and clustering entities between multiple (>2) sources either treated the different sources as a single source or assumed that the individual sources are duplicate-free, so that only matches between sources have to be found. In this work we propose and evaluate a general Multi-Source Clean Dirty (MSCD) scheme with an arbitrary combination of clean (duplicate-free) and dirty sources. For this purpose, we extend a constraint-based clustering algorithm called Affinity Propagation (AP) for entity clustering with clean and dirty sources (MSCD-AP). We also consider a hierarchical version of it for improved scalability. Our evaluation considers a full range of datasets containing 0% to 100% of clean sources. We compare our proposed algorithms with other clustering schemes in terms of both match quality and runtime. The proposed algorithms outperform previous methods and achieve an excellent precision in MSCD scenarios.

Keywords: Entity Resolution; Clustering; Affinity Propagation; MSCD-AP

1 Introduction

Entity Resolution (ER), also referred to as record linkage or deduplication, is a main data integration task. It is used to identify entities, such as specific customer or product descriptions, in one or several data sources that refer to the same real-world entity. Most previous ER approaches focus on finding such matches in either a single source or between two sources. Multi-source ER aims at finding matching entities in an arbitrary number of sources which is more challenging than dealing with 1-2 sources since not only the degree of heterogeneity but also the variance in data quality generally increases with the number of sources.

There are two main phases for multi-source ER [Ra16, Sa18, Ch19]. First, similar pairs of entities are determined over all sources as match candidates. These can be recorded in a similarity graph where each vertex represents an entity and each edge a match relationship between two entities. Edges may have a similarity score reflecting the match probability. In the second phase, the matches are determined by a clustering algorithm on the similarity graph. All matching entities from any source referring to the same real-world entity are

¹ University of Leipzig & ScaDS.AI Dresden/Leipzig, s.lerm@studserv.uni-leipzig.de

² University of Leipzig & ScaDS.AI Dresden/Leipzig, saeedi@informatik.uni-leipzig.de

³ University of Leipzig & ScaDS.AI Dresden/Leipzig, rahm@informatik.uni-leipzig.de

grouped in one cluster. There are many possible approaches for this entity clustering, especially the ones that have been proposed for clustering matches in a single source [Ha09, SPR17]. In the special case of duplicate-free (clean) sources each cluster contains at most one entity per source so that the cluster size is limited by the number of sources. Cluster algorithms that utilize this restriction have been shown to achieve better match quality than the more general approaches [NGR16, SPR18].

In this paper, we investigate a Multi-Source Clean Dirty (MSCD) entity clustering approach that can utilize clean sources but can also deal with dirty sources so that only a fraction (possibly 0%) of the sources have to be clean. The goal is to achieve better match quality than with a general clustering scheme when there are clean sources while avoiding the limitation of requiring that all sources have to be clean. While one could first deduplicate dirty sources and then apply a clustering for clean sources, the effort to determine these source-specific deduplication approaches is immense and perhaps not completely successful. We experimented with such an approach for a data integration challenge [OSR19] but it performed worse than matching dirty sources. Consequently, it is more flexible to support a mix of both dirty and clean sources. For this purpose, we propose extensions to the Affinity Propagation (AP) clustering approach [FD07] that converts the problem of clustering into a constraint optimization problem. Our extension MSCD-AP adds a new constraint to AP to deal with clean sources. We also consider a hierarchical variation of MSCD-AP for improved scalability, provide parallel implementations based on Apache Flink and comparatively evaluate the new approaches.

We make the following contributions:

- We are the first to consider a mix of clean and dirty sources for multi-source entity resolution and propose an extended version of affinity propagation clustering, MSCD-AP, for this purpose.
- For improved scalability, we propose a hierarchical variation, MSCD-HAP, and provide parallel implementations for the clustering schemes based on Apache Flink.
- We perform a comprehensive evaluation of the match quality, runtimes and scalability of the new approaches for different datasets and compare them with previous clustering schemes.

After a discussion of related work, we give a brief summary of the standard AP algorithm in Section 3. Section 4 presents the new clustering method MSCD-AP in detail while Section 5 describes the scalable approach MSCD-HAP. In Section 6 we present our evaluation.

2 Related Work

Entity resolution has been the subject of a large amount of research as can be seen from many surveys and books such as [Ch12, Ch19, KR10, GM12, Pa19]. For larger datasets it is imperative to apply blocking techniques to reduce the number of comparisons of entity pairs.

There are also many ways to determine match candidates, e.g. match rules requiring that a combined similarity of selected attributes exceed some threshold or supervised approaches using training data with both matching and non-matching pairs of entities to determine a match classifier.

This paper focuses on the final step of the ER pipeline, entity clustering on a similarity graph, to group together all matches of a real-world object. Clustering can improve match quality over the binary links in the similarity graph as it is possible to transitively infer additional links or to eliminate links that are unlikely to be correct. There are numerous approaches for clustering and also for entity clustering. Most previous entity clustering approaches focus on finding matches in a single (dirty) source. Example approaches include Connected Components, Center and Merge-Center clustering [HM09], Affinity Propagation [FD07], Ricochet clustering [WB09], Markov clustering [VD00] and Correlation clustering [BBC04]. [Ha09] comparatively evaluated many of these algorithms for a single source.

In [SPR17] we have shown that these approaches can be adapted for multi-source entity clustering and we comparatively evaluated several approaches for such a setting. We further developed new multi-source entity clustering approaches such as CLIP [SPR18], that work for clean (duplicate-free) data sources and can outperform the more general approaches for dirty sources. In our evaluation we will compare the new MSCD entity clustering approaches based on affinity propagation with these previous methods for dirty and clean sources. The previous clustering approaches including CLIP have been integrated into the FAMER⁴ framework [Sa18] for multi-source entity resolution, that is used for our comparative evaluation. All match and clustering approaches in FAMER are implemented on top of Apache Flink to achieve a parallel entity resolution on a cluster of machines in order to reduce runtimes and improve scalability to larger datasets.

3 Affinity Propagation Clustering

The Affinity Propagation clustering algorithm [FD07] groups entities by identifying exemplars. An exemplar is the entity that best represents all the entities of a cluster. The non-exemplar entities are assigned to the most appropriate exemplar. The goal of AP is to find exemplars and cluster assignments in a way that the sum of similarities inside clusters are maximized.

In [GF09], AP is solved by the iterative max-sum algorithm on a factor graph. The factor graph is a bipartite graph between the exemplar assignments (variable nodes) and factor nodes representing two constraints, called the g - and h -constraints. Figure 1a illustrates such a factor graph for AP. Variable nodes and factor nodes are represented as circles and rectangles respectively. For clustering n entities, the factor graph is represented by a n^2 binary matrix \mathbb{B} . The variable b_{ij} has the value 1 if the datapoint (entity) j is the exemplar

⁴ https://dbs.uni-leipzig.de/research/projects/object_matching/famer

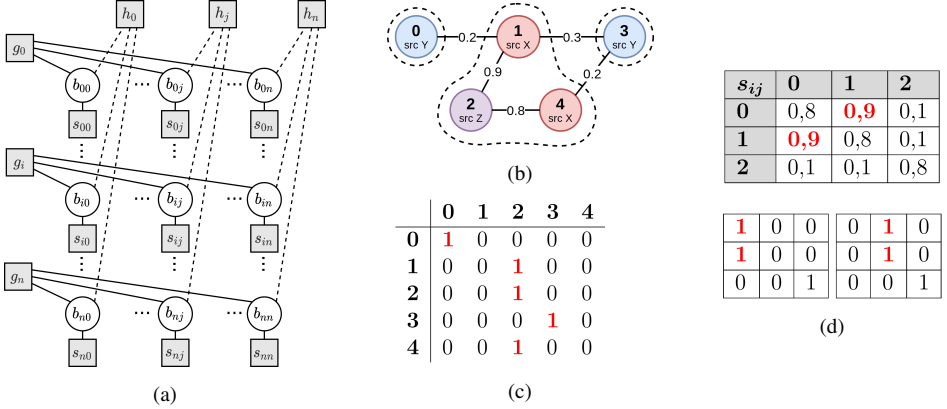


Fig. 1: a) Factor graph of AP [AKK19] b) AP clustering example c) Binary matrix d) Oscillation

of i . The factor nodes g_i and h_j assure a valid clustering by applying the constraints. The g -constraint enforces that a datapoint has to have exactly one exemplar. It means in each row of the binary matrix there must be exactly one variable with value 1. The h -constraint assures that a datapoint selects itself as its exemplar, if it is already chosen as exemplar by at least one other datapoint. It means, if there exists at least one 1 in a column of the binary matrix, then the diagonal element b_{jj} of that column must be set to 1 too. The cluster assignments are based on the similarities between entities so that similarity values are also represented as factor nodes (factor node s_{ij} provides the similarity information between the entities i and j).

Figure 1b illustrates an example clustering of AP where five entities 0-4 from three (differently colored) sources X, Y and Z are grouped in three clusters. The corresponding output binary matrix in Figure 1c shows that entities 0, 2 and 3 are the exemplars of the three clusters. As described above, the rows of the binary matrix illustrate the exemplar (cluster) assignment while the columns depict the clusters. The group of 1 values in column j represents the entities of the cluster with exemplar j .

AP aims at finding a cluster assignment maximizing the sum of similarities within clusters. This optimization problem can be formulated with the energy function [AKK19] shown in Equation (1). Maximizing the function requires to find an optimal configuration of the variables in \mathbb{B} so that the sum of the similarities between entities and their exemplars is maximized and the two constraints are met. An exact maximization of the energy function is computationally intractable because a special case of this maximization problem is the NP-hard k-median problem [FD07].

$$E(\mathbb{B}) = \sum_{ij} s_{ij} b_{ij} + \sum_i g_i(\mathbb{B}(i, :)) + \sum_j h_j(\mathbb{B}(:, j)) \quad (1)$$

with

$$g_i(\mathbb{B}(i, :)) = \begin{cases} 0 & \text{if } \sum_j b_{ij} = 1 \\ -\infty & \text{otherwise} \end{cases} \quad h_j(\mathbb{B}(:, j)) = \begin{cases} 0 & \text{if } b_{jj} = \max_i b_{ij} \\ -\infty & \text{otherwise} \end{cases}$$

The proposed iterative max-sum algorithm uses several parameters that affect the clustering result and that deal with the problem of non-convergence. The most important parameter is called *preference*. It defines the self-similarity s_{ii} of an entity i . The higher the preference value is chosen the more likely the entity becomes an exemplar. Parameters to deal with non-convergence are the noise level and the damping factor λ . AP suffers from oscillation between solutions that are similarly well suited for optimizing the energy function. For the similarity matrix in the top portion of Figure 1d, the symmetrical similarity values between entities 0 and 1 make both equally well suited as an exemplar. In such a situation, AP does not converge and oscillates between the two solutions with either entity 0 or 1 as the exemplar as shown in the bottom part of Figure 1d. Oscillation is avoided by adding a tiny amount of noise to the similarity values. The damping factor has a similar goal and is related to the used message passing implementation for the iterative computation. It leads to an adaptation of values exchanged between iterations. If oscillations nevertheless occur, the preference or the damping factor must be adapted (see next section).

4 MSCD Affinity Propagation

To cluster mixed datasets of clean and dirty sources, we propose an extension to AP called MSCD-AP. Since clean sources have no duplicates, every cluster should have at most one entity of a clean source. This is now controlled by an additional *clean-source consistency* constraint. It means that in each column of the binary assignment matrix \mathbb{B} , value 1 is allowed for at most one (row) entity of a clean source.

Figure 2a shows a possible clustering of MSCD-AP for the running example when sources X and Y are clean. There are four *source-consistent* clusters with at most one entity per clean source. In the corresponding binary matrix, each column has at most one entity with value 1 per clean source. For example, the column (cluster) for exemplar entity 1 has two associated entities (1 and 2) from different sources.

Our proposed clean-source consistency constraint is expressed in Equation (2). It uses function t to add a large penalty to the extended energy function in Equation (3) when the constraint is violated. The constraint requires that for a column j the value 1 is allowed for at most one datapoint from a clean source Q .

$$t_{Qj}(\mathbb{B}(i \in Q, j)) = \begin{cases} 0 & \text{if } \sum_{i \in Q} b_{ij} \leq 1 \\ -\infty & \text{otherwise} \end{cases} \quad (2)$$

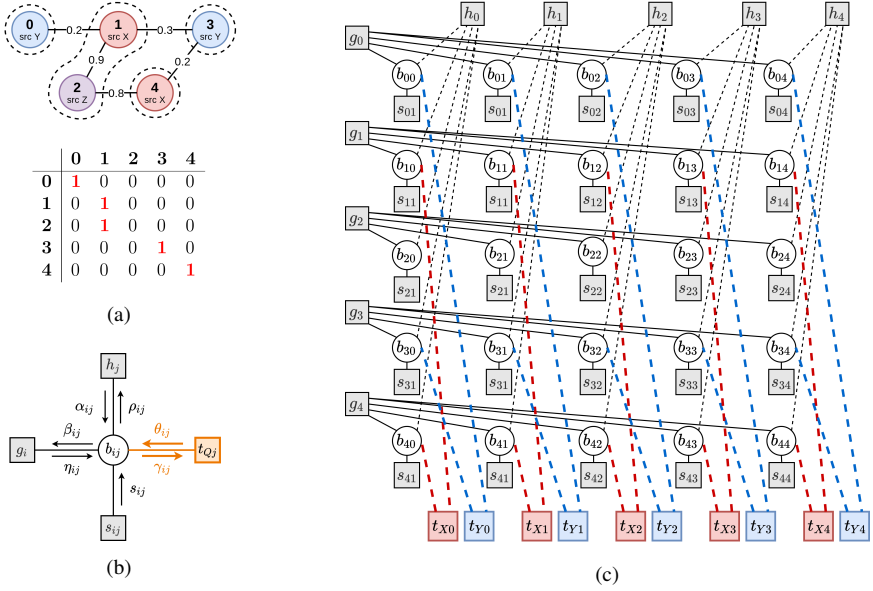


Fig. 2: a) MSCD-AP clustering example b) Messages of the MSCD-AP factor graph c) The factor graph for MSCD-AP for the running example. The sources X and Y are clean.

$$E(\mathbb{B}) = \sum_{ij} s_{ij} b_{ij} + \sum_i g_i(\mathbb{B}(i, :)) + \sum_j h_j(\mathbb{B}(:, j)) + \sum_Q \sum_{i \in Q, j} t_{Qj}(\mathbb{B}(i, j)) \quad (3)$$

Figure 2c illustrates the extension of the AP factor graph to cluster our running example data. For clean sources X and Y, additional factor nodes t_x and t_y (marked in red and blue) are added to each column of the binary matrix. The factor node t_{xj} assures the clean-source consistency constraint for source X and column j . It is connected to the variable node b_{ij} only if entity i is from data source X. The clean-source constraint may get in conflict with the h -constraint of AP. The h -constraint enforces a datapoint to choose itself as its own exemplar, if it is selected by at least one other datapoint. So the diagonal element b_{jj} of column j is enforced to be 1, if there is any other 1 in that column. On the other hand, the clean-source constraint enforces b_{jj} to be 0, if another datapoint of the same clean source selected it as its exemplar. So the two constraints enforce different values for b_{jj} and thus the algorithm may struggle to converge. This situation is simply avoided in our implementation by not having links between entities of the same clean source which is a default feature of the linking component of FAMER.

For the traditional AP clustering, the max-sum optimization has been implemented by a message passing algorithm [GF09]. The messages are exchanged between factor and variable nodes of the factor graph to reflect the mutual dependencies within an iterative

process. The messages are computed differently depending on whether the recipient node is a variable node or a factor node. Figure 2b shows the messages exchanged between the nodes of the new factor graph of MSCD-AP. The grey-colored factor nodes enforce the g and h constraints while the new factor node t (marked in orange) applies the clean-source consistency constraint via the θ and γ messages.

We build on the formulae from [Gi12] to update messages for the original constraints and specify the new message formulas for our MSCD extension. In the max-sum algorithm, outgoing messages of a variable node summarize all incoming messages to that node, except of the node to which the new message will be sent. Due to the new constraint, all outgoing messages from variable nodes to factor nodes are now modified because the new factor nodes t_{Qj} are additional neighbours of b_{ij} . As sum of the incoming messages from the neighbouring nodes, except of the recipient, the modified messages β and ρ as well as the new message γ are easily deduced as listed in Equation (4) - (6).

The message formulas from factor nodes to variable nodes do not change in AP when a new factor node is added. Therefore the incoming messages of α (eq. (7)) and η (eq. (8)) remain unchanged compared to AP. The new incoming message θ from the new factor node t_{Qj} is expressed in Equation (9). The more complex derivation of message θ from the max-sum algorithm is given in the appendix. The variable assignments that maximize the energy function are calculated by Equation (10).

$$\beta_{ij} = s_{ij} + \alpha_{ij} + \theta_{ij} \quad (4) \quad \rho_{ij} = s_{ij} + \eta_{ij} + \theta_{ij} \quad (5) \quad \gamma_{ij} = s_{ij} + \alpha_{ij} + \eta_{ij} \quad (6)$$

$$\alpha_{ij} = \begin{cases} \sum_{k \neq j} \max(0, \rho_{kj}) & i = j \\ \min[0, \rho_{jj} + \sum_{k \neq \{i, j\}} \max(0, \rho_{kj})] & i \neq j \end{cases} \quad (7)$$

$$\eta_{ij} = -\max_{k \neq j} \beta_{ik} \quad (8) \quad \theta_{ij} = \min(0, -\max_{k \neq i} [\gamma_{kj}]) \quad (9)$$

$$b_{ij} = \begin{cases} 1 & \alpha_{ij} + \rho_{ij} > 0 \\ 0 & \alpha_{ij} + \rho_{ij} \leq 0 \end{cases} \quad (10)$$

Algorithm 1 lists the pseudo code of MSCD-AP with focus on the parameter adaptation. There are several inputs for the algorithm. The clustering problem is defined by the similarity matrix S and the specification of the clean sources (*srcInfo*). λ denotes the damping factor. The preference can be set separately for dirty (p_{dirty}) and clean (p_{clean}) sources. Random gaussian noise is added to the similarity values at a decimal position specified by the *noiseLevel*. Parameter adaption for the preference values and the damping factor is done stepwise by $step_{pref}$ and $step_{dmp}$. The adaptation steps are real values in $(0, 1]$ that are used to increase the original values towards the maximum 1 or decrease them towards 0. As algorithm output the binary matrix \mathbb{B} describes the exemplar assignment of every entity.

Algorithm 1: MSCD-AP**Input:** S , $srcInfo$, λ , p_{dirty} , p_{clean} , $noiseLevel$, $step_{pref}$, $step_{dmp}$ **Result:** \mathbb{B} with exemplar assignments

```

1 repeat
2    $initializeMessages()$ ;
3    $initializeB()$ ;
4    $modifyS(p_{dirty}, p_{clean}, noiseLevel, srcInfo)$ ;
5   for  $iteration = 0 : max$  do
6      $updateMessages(\lambda)$ ;
7      $updateB()$ ;
8     if  $isConverged()$  then  $break$ ;
9      $solutionFound \leftarrow isSolutionFound(\mathbb{B})$ ;
10    if  $\neg solutionFound$  then  $adaptParameters(step_{pref}, step_{dmp})$ ;
11 until  $solutionFound$ ;
```

After the initialization of the messages and output matrix (line 2 and 3) the diagonal elements s_{jj} of the similarity matrix are set to the defined preference values and noise is added to all similarity values in line 4. The iterative message passing starts in line 5. In each iteration, the messages are updated in line 6 according to Equation (4) - (8). Additionally, α and ρ messages are damped in order to prevent oscillations. Finally in line 7, the binary matrix values are updated according to Equation (10). If no changes are observed in the binary matrix after a specific number of iterations, the algorithm converges and is ended (line 8). Otherwise it ends after a maximal number of iterations. If the algorithm stops but the solution is not found yet (line 9 and 10), then it has to be restarted with adapted parameters. For this purpose, function $adaptParameters$ initially decreases the preference values by *preference adaption step* ($step_{pref}$) until the minimum value 0. If convergence is still not reached, the preference values are then increased step by step until the maximum 1 is reached. In case of no success, the preference values are reset to their original values and the damping factor λ is now increased by *damping adaption step* ($step_{dmp}$). This process continues until the algorithm finds a valid solution.

5 Scalable MSCD Affinity Propagation

Clustering large datasets is a challenge for AP since its time and memory complexity grows quadratically with the number of entities and thus the data volume⁵. Liu et al. [Li13] proposed Hierarchical Affinity Propagation (HAP) to make AP suitable for clustering large-scale datasets. Following a divide and conquer strategy, HAP clusters the dataset by executing AP several times on different hierarchy levels.

⁵ In the case of a sparse similarity matrix, the time complexity reduces to $Nk \log(N)$ with k being the average connectivity of the similarity matrix [Zh10].

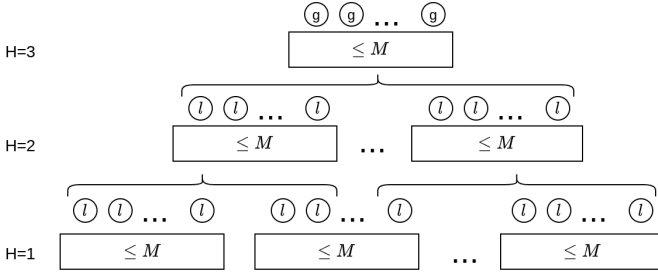


Fig. 3: HAP for three hierarchy levels H . Circles illustrate local (l) and global (g) exemplars, rectangles represent partitions.

Figure 3 illustrates the hierarchical clustering for three levels. In the first (lowest) hierarchy level, the dataset is randomly divided into equal-sized partitions of maximal size M . Then AP is executed on each partition, resulting into a set of so called *local exemplars* for each partition. In the next hierarchy level, the exemplars of the previous level are merged and again partitioned. This process is repeated until the input size of a hierarchy level is lower or equal to M . The execution of AP on the top hierarchy level determines the *global exemplars* for the dataset. All non-exemplar entities are assigned to the global exemplar with the highest similarity. Thus AP is executed once for each partition of each hierarchy level with a complexity of $O(M^2)$.

Unfortunately, applying the hierarchical algorithm for MSCD-AP does not guarantee the clean-source consistency. This is because, the clustering of local exemplars by MSCD-AP on intermediate hierarchy levels violates the clean-source consistency when two local exemplars from a previous level are clustered together although they have associated entities from the same clean source. A naive solution is to extend each local exemplar with the source information of the entities assigned to it in the previous hierarchy level. This could be used in subsequent cluster decisions to avoid that more than one entity of a clean source is assigned to an exemplar. This approach, however, can lead to poor clustering results. A bad decision in a lower level of the hierarchy, where an entity of a clean source with a low similarity is assigned to a local exemplar, can prevent that a much more similar entity from the respective source is merged at a higher level resulting in poor cluster decisions.

A more promising solution is to assign entities to global exemplars separately for clean and dirty sources. Initially, HAP is executed using MSCD-AP to determine local and global exemplars on the partitions. As in HAP, dirty source entities are then assigned to the exemplars with the highest similarity. By contrast, clean source entities are assigned using the Hungarian algorithm [Ku55, Mu57]. Given the similarities between these entities and exemplars, the Hungarian algorithm finds a 1:1 assignment between entities of a clean source and exemplars (i.e., each exemplar is assigned to at most one entity of a clean source) so that the overall similarity of all assignments is maximized. If the number of entities from a clean source exceeds the number of exemplars, the excess points form singleton

Tab. 1: Overview of evaluation datasets

General information						Perfect result	
domain	entity properties		#entity	#src	type	#clusters	#links
DS-G	geography	label, longitude, latitude	3,054	4	MSC	820	4,391
DS-M	music	artist, title, album, year, length	19,375	5	MSC	10,000	16,250
DS-P1	persons	name, surname, suburb, postcode	5,000,000	5	MSC	3,500,840	3,331,384
DS-P2			10,000,000	10	MSC	6,625,848	14,995,973
DS-C	camera	heterogenous key-value pairs	21,023	23	MSCD	3,910	368,546

clusters. When a global exemplar is from a clean source, the clean-source consistency is also enforced since there is no similarity link between entities of the same clean source.

The Hungarian algorithm has a computational complexity of $O(mk^2)$ for a $m \times k$ cost matrix [Cu16] with k global exemplars and m entities from one clean source. The complexity is higher compared to AP, but the bipartite matching is executed on small subsets of the n -sized dataset ($m, k \ll n$). Thus the combination of HAP with MSCD-AP and the Hungarian algorithm is still more suitable for large datasets than MSCD-AP. We call this combination MSCD-HAP and comparatively evaluate it in the next section.

6 Evaluation

We now evaluate the cluster effectiveness and efficiency of the proposed MSCD extensions of AP in comparison to standard AP and previous clustering schemes. We first describe the used datasets from four domains. We then analyze comparatively the effectiveness of the proposed algorithm. Finally, we evaluate runtime performance and scalability.

6.1 Datasets and Configuration Setup

We evaluate the new approaches with four multi-source datasets of clean sources (MSC) that have also been used in previous studies [SPR17, SPR18, Sa18]. Table 1 gives an overview of the datasets from three domains (geography, music, persons) including available properties and number of entities. For the evaluation of mixed datasets of clean and dirty sources, we use the dataset of the ACM SIGMOD 2020 Programming Contest⁶. It contains approximately 30k product specifications from 24 dirty sources. For our purposes, we determine a subset called DS-C focussing on camera products (Table 1). We excluded the source *www.alibaba.com* because it contains just a few cameras but many non-cameras. Table 2 lists the 23 remaining sources and their number of entities with and without duplicates. The matching result of the SIGMOD contest winner [BI20] is considered as the ground truth. It achieved f-measure of 99% by extensive domain-specific preprocessing and

⁶ <http://www.inf.uniroma3.it/db/sigmod2020contest/index.html>

Tab. 2: Overview of camera dataset (DS-C)

Source name	ID	#entity	#entity dedup.
buy.net	1	358	244
cammarkt.com	2	198	94
www.buzzillions.com	3	832	630
www.cambuy.com.au	4	118	56
www.camerafarm.com.au	5	120	59
www.canon-europe.com	6	164	163
www.ebay.com	7	14,009	3,255
www.eglobalcentral.co.uk	8	190	75
www.flipkart.com	9	118	47
www.garricks.com.au	10	130	69
www.gosale.com	11	895	578
www.henrys.com	12	181	137
www.ilgs.net	13	102	64
www.mypcconnection.com	14	347	279
www.pccconnection.com	15	211	126
www.pricer-hunt.com	16	327	282
www.pricedekho.com	17	366	325
www.priceme.co.nz	18	740	475
www.shopbot.com.au	19	516	334
www.shopmania.in	20	630	556
www.ukdigitalcameras.co.uk	21	129	73
www.walmart.com	22	195	115
www.wexphotographic.com	23	147	87
sum		21,023	8,123

Tab. 3: MSCD datasets

Name	%cln ¹	cln ²	#cln ³	#dirt ⁴
DS-C0	0		0	21,023
DS-C26	26	1-6, 8-23	4,868	14,009
DS-C32	32	7	3,255	7,014
DS-C50	50	7, 18, 19, 20, 22, 23	4,822	4,786
DS-C62A	62	1, 4, 6, 7, 9, 11, 13, 15, 17, 19, 20	5,748	3,536
DS-C62B	62	2, 3, 5, 7, 8, 10, 12, 14, 16, 18, 21-23	5,630	3,478
DS-C80	80	1-12, 14-18	6,894	1,719
DS-C100	100	1-23	8,123	0

¹ Percentage of entities from clean sources² Clean source IDs³ Number of entities from clean sources⁴ Number of entities from dirty sources

matching camera entities against a prepared list of nearly all available cameras in the market. Our matching and clustering approaches are generic and applicable to different datasets. Our goal is not to achieve the best possible result but to enable a fair comparison of the clustering schemes based on reasonably good input similarity graphs for different datasets.

Using DS-C, we create eight datasets with different combinations of clean and dirty sources and thus different degrees of dirtiness. As shown in Table 3, we name the datasets according to the percentage of entities from clean sources, where DS-C0 and DS-C100 means that all entities are from dirty and clean sources, respectively. For the mixed cases, an important distinction is whether a clean or dirty version of source 7 (*www.ebay.com*) is considered because it is the largest source and contains many duplicates. In DS-C62A and DS-C62B, the clean form of source 7 is included, while all other sources that are clean in 62A are dirty in 62B and vice versa.

Tab. 4: Linking configurations of clean multi-source datasets

Blocking Key		Similarity Function
DS-G	prefixLength1 (label)	Jaro-Winkler (label) & geographical distance
DS-M	prefixLength1 (album)	Trigram (title)
DS-P1/P2	prefixLength3 (surname) + prefixLength3 (name)	avg (Trigram (name) + Trigram (surname) + Trigram (postcode) + Trigram (suburb))

The blocking and matching configurations for the clean datasets are listed in Table 4 and correspond to the ones in previous studies [SPR18, Sa18]. For the camera dataset, we extracted the manufacturer name, a list of model names, manufacturer part number (mpn), european article number (ean), digital and optical zoom, camera dimensions, weight, product code, sensor type, price and resolution from the heterogeneous product specifications. In order to reduce the number of comparisons, standard blocking with a combined key of manufacturer name and model number is applied. Within these blocks, all pairs with exactly the same model name, mpn or ean are classified as matches. We assign a similarity value to the matched pairs determined from a weighted average of the character-3Gram Dice similarity of string values and a numerical similarity of numerical values (within a maximal distance of 30%).

6.2 ER Quality of Clustering Algorithms

To evaluate the quality of the clustering results, we use the standard metrics precision, recall and their harmonic mean, f-measure w.r.t. the links of the perfect cluster results (last column of table 1). We compare the quality of AP and the proposed MSCD-AP approaches with seven previous clustering schemes comprised in FAMER [Sa18]. The CLIP approach is tailored to clean sources. The other six algorithms are general approaches for dirty sources (connected components, correlation clustering CCPivot, two variants of star clustering and two variants of center clustering). We also provide the quality of the input similarity graph (without clustering) in our figures. For AP and MSCD-AP we manually determined suitable parameter configurations. We use the interval $[0.01, 0.7]$ for preference values and set a higher preference value for clean sources than for dirty sources to choose exemplars preferably from clean sources. The damping factor is set to 0.5 and noise is added to the similarity values from the third decimal place. For the smaller datasets DS-G, DS-M and DS-C, we used a partition size of 1000 while for the person datasets we apply MSCD-HAP with partition size 100 to reduce runtimes. When the size of a connected component is smaller than the partition size, MSCD-AP is executed. Higher similarity thresholds result in fewer links and smaller components that are mostly executed without partitioning. Because DS-G and DS-C consist of small components, partitioning is not used. Thus AP and MSCD-AP are executed. In DS-P the hierarchical algorithms are used. For reasons of space the results of DS-P1 are omitted, as they are very similar to those of DS-P2.

We first analyze cluster quality for the datasets with only clean sources. Figure 4 shows the results for the three MSC datasets for different similarity thresholds to generate the input similarity graph. As expected, the f-measure results are best for the CLIP approach tailored to ER for clean sources. However, the proposed MSCD-AP approach achieves about the same quality for two datasets (DS-G, DS-P2) and performs better than the six general clustering schemes for DS-M. It also outperforms AP in all cases. These surprisingly good results are mainly due to an excellent precision of MSCD-AP which can outweigh its comparatively low recall. The recall is limited since AP and MSCD-AP strongly depend

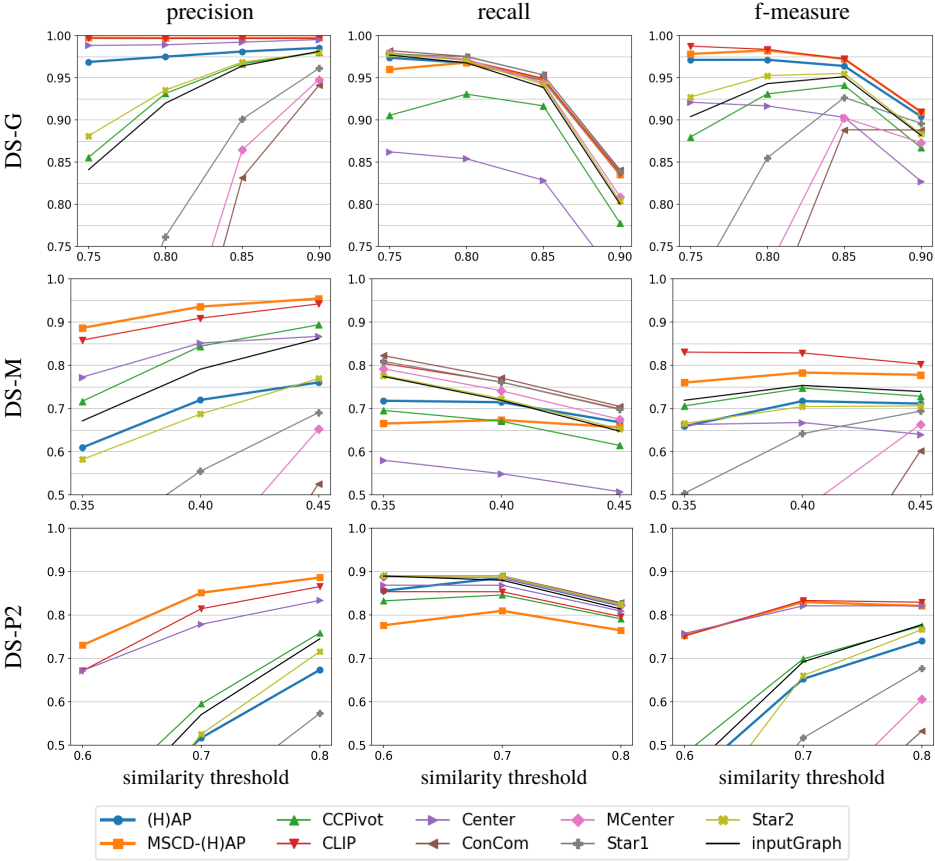


Fig. 4: Clustering quality for the multi-source clean ER datasets

on the relative similarity values and can even consider a high similarity value such as 0.8 as low if it is below the average of the considered value range, e. g. [0.8, 1.0]. This leads to more small clusters and thus a lower recall compared to other algorithms. Due to the clean-source constraint, MSCD-AP creates more exemplars than AP and therefore obtains a lower recall compared to AP but a much better precision.

Figure 5 shows the quality of the clustering results for the camera datasets with different degrees of dirtiness. Due to space constraints we show results for 5 of the 8 cases but the results for the remaining datasets confirm the overall outcome. We observe that MSCD-AP achieves the best f-measure for all cases with a mix of dirty and clean sources. For the case of only clean sources (DS-C100) it is only outperformed by CLIP. For dirty sources only (DS-C0) MSCD-AP is identical to AP which is among the best approaches. As a result, MSCD-AP is the best or among the best approaches over all configurations while other

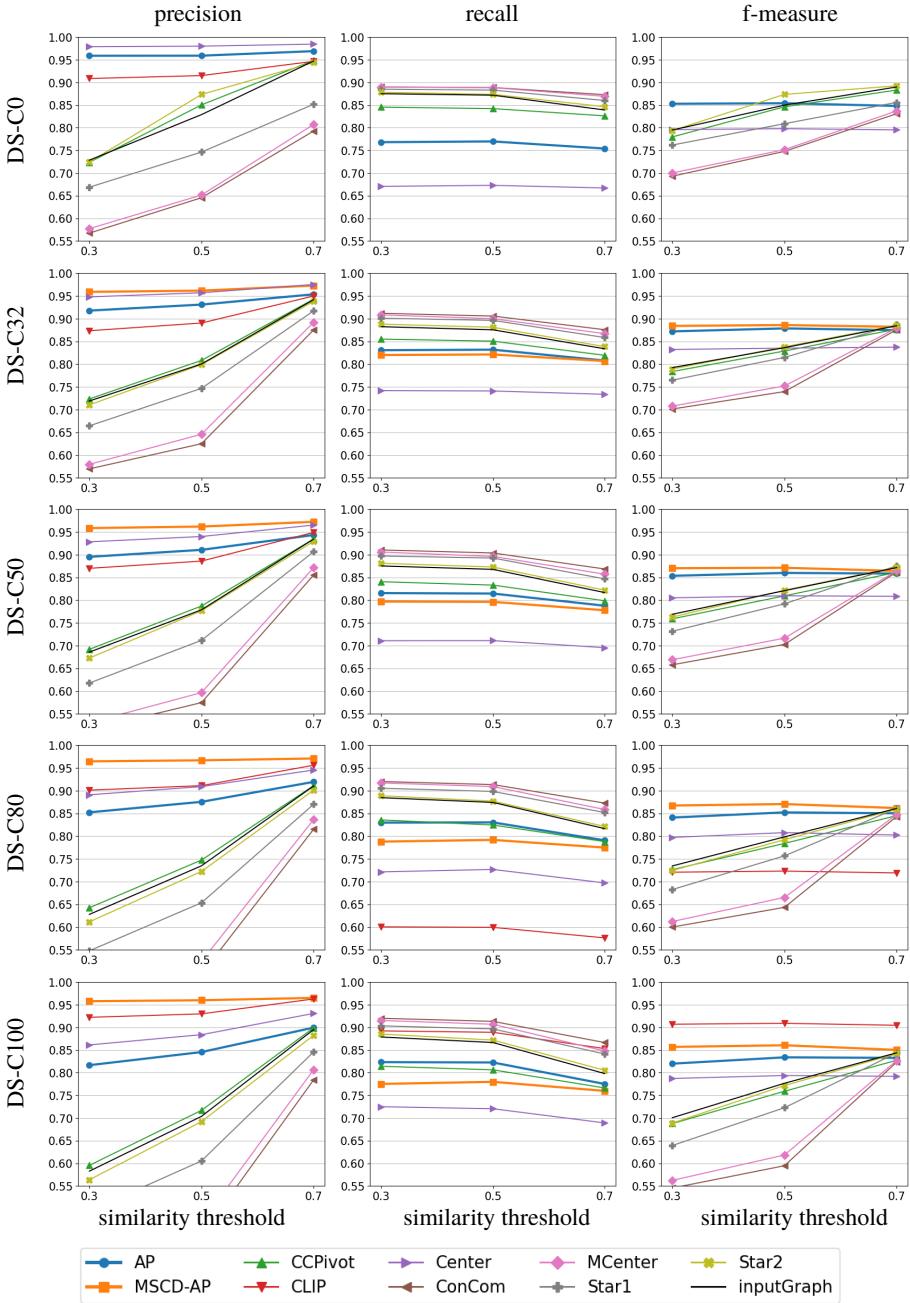


Fig. 5: Clustering quality for the multi-source clean/dirty ER datasets

schemes like CLIP are good in only one configuration. Another strong point of MSCD-AP is that its f-measure is nearly stable over all threshold values used to determine the input similarity graph while the general clustering schemes depend on finding a suitable threshold. As for the MSC datasets, the good results of MSCD-AP are mainly due to its excellent precision values in all cases that outweigh its lower recall results.

6.3 Runtimes and Speedups

We evaluate runtimes and speedup behavior for the larger datasets from the person domain. The speedup of MSCD-HAP is determined for the parallel execution with different numbers of workers. We also analyze the effect of different MSCD-HAP partition sizes on runtime as well as on clustering quality. The experiments are performed on a shared nothing 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 1.9.0. We run Apache Flink standalone with 6 threads and 40 GB memory per worker.

Figure 6 shows the runtime of each clustering approach for a parallel execution on 16 workers. As expected, the larger dataset DS-P2 leads to higher runtimes than for DS-P1 while higher similarity thresholds reduce runtimes due to the lower number of edges in the similarity graph. MSCD-HAP is slower than HAP because the calculations for the clean-source constraint and the exemplar assignment by the Hungarian algorithm need additional runtime. The clean-source constraint of MSCD-AP also leads to more exemplars and potentially more entities of the same source that are equally well suited to be an exemplar. Thus, oscillations occur more frequently for MSCD-AP compared to AP leading to more parameter adaptations to find a converging solution.

MSCD-HAP along with CCPivot and MergeCenter are among the slowest algorithms for the lowest threshold. Yet with higher similarity thresholds the runtime of MSCD-HAP

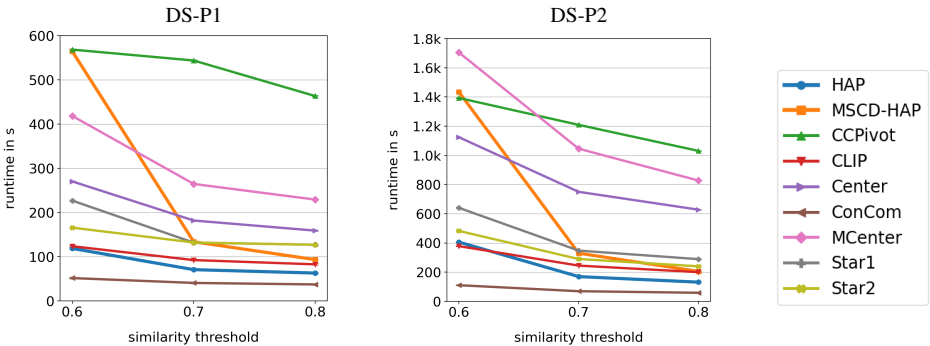


Fig. 6: Runtimes for clustering schemes (with partition size 100 for HAP and MSCD-HAP)

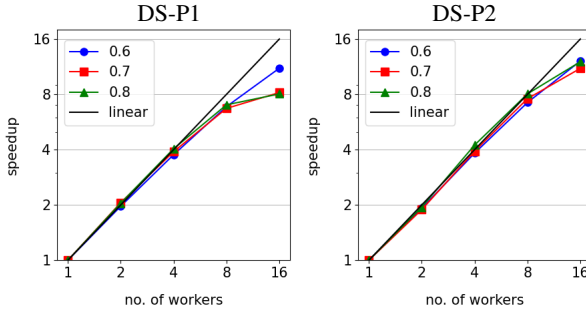


Fig. 7: Speedup of MSCD-HAP for different similarity thresholds

improves significantly making it one of the fastest algorithms. This is because a high minimum threshold avoids that a large number of entities are connected in the similarity graphs resulting in mostly small clusters and reduced work for the Hungarian algorithm. Moreover, oscillations occur less in such cases.

Figure 7 depicts the speedup of MSCD-HAP with partition size 100 for different similarity thresholds and for 1 to 16 worker machines. We observe that close to perfect speedup is achieved for the larger dataset DS-P2 and for a lower similarity threshold (bigger similarity graph) for the smaller DS-P1 dataset. For the higher thresholds the needed computations for DS-P1 cannot utilize 16 machines so that a good speedup is only achieved until 8 workers.

Figure 8 investigates the effect of partition size on both runtime and clustering quality.

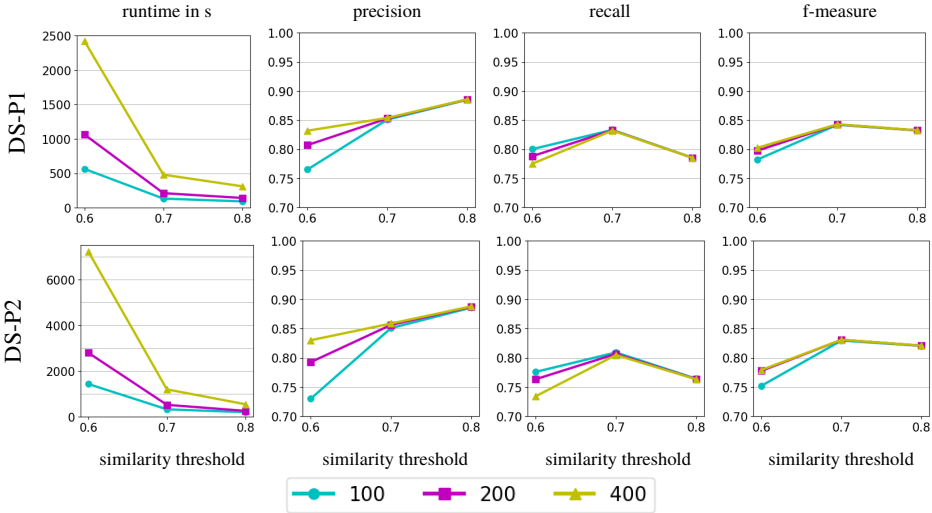


Fig. 8: Clustering quality and runtime for different partitions sizes of MSCD-HAP

We observe that larger partition sizes lead to much higher runtimes but also to improved clustering quality. These effects are most pronounced for smaller similarity threshold such as 0.6 that lead to bigger similarity graphs and thus to more computations. With larger partition sizes there are more entities and more similarity values in each partition. Therefore, the probability of finding good local and global exemplars rises and consequently the precision is improved. Yet recall drops slightly, because on bigger partitions more exemplars can be found and AP generally tends to form many small clusters. While the runtime is up to seven times higher for partition size 400 compared to 100 (for DS-P2) for threshold 0.6, these differences largely go away for higher thresholds and much smaller similarity graphs. This is also the case for clustering quality, where similarity value 0.7 or higher leads to about the same f-measure for all partition sizes.

7 Conclusions

We studied how to support multi-source entity clustering for a mix of clean (duplicate-free) and dirty data sources. The proposed extension of Affinity Propagation clustering, MSCD-AP, showed to be highly effective and perform better than previous methods for mixed configuration where a subset of the sources is duplicate-free. To improve runtimes we proposed the use of a hierarchical version MSCD-HAP and provide parallel implementations of the algorithms. The parallel implementations achieve good speedup values thereby supporting scalability to larger datasets. In future work, we will investigate how to extend additional clustering schemes for multi-source ER for mixed configurations with both clean and dirty sources.

8 Acknowledgements

This work is partially funded by the German Federal Ministry of Education and Research under grant BMBF 01IS18026B in project ScaDS.AI Dresden/Leipzig.

Bibliography

- [AKK19] Amjad, R.; Khan, R.; Kleinsteuber, M.: Extended Affinity Propagation: Global Discovery and Local Insights. *IEEE Transactions on Knowledge & Data Engineering*, 2019.
- [BBC04] Bansal, N.; Blum, A.; Chawla, S.: Correlation clustering. *Machine learning*, 56(1-3):89–113, 2004.
- [BI20] Blacher, M.; Klaus, J.; Mitterreiter, M.; Giesen, J.; Laue, S.: Fast Entity Resolution With Mock Labels and Sorted Integer Sets. *CEUR Workshop Proceedings*, 2726, 2020.
- [Ch12] Christen, P.: Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer Science & Business Media, 2012.

- [Ch19] Christophides, V.; Efthymiou, V.; Palpanas, T.; Papadakis, G.; Stefanidis, K.: End-to-End Entity Resolution for Big Data: A Survey. *arXiv preprint arXiv:1905.06397*, 2019.
- [Cu16] Cui, H.; Zhang, J.; Cui, C.; Chen, Q.: Solving large-scale assignment problems by Kuhn-Munkres algorithm. 2nd International Conference on Advances in Mechanical Engineering and Industrial Informatics (AMEII), 2016.
- [FD07] Frey, B. J.; Dueck, D.: Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.
- [GF09] Givoni, I. E.; Frey, B. J.: A binary variable model for affinity propagation. *Neural computation*, 21(6):1589–1600, 2009.
- [Gi12] Givoni, I. E.: Beyond affinity propagation: Message passing algorithms for clustering. *CiteSeer*, 2012.
- [GM12] Getoor, L.; Machanavajjhala, A.: Entity resolution: theory, practice & open challenges. *Proceedings of the VLDB Endowment*, 5(12):2018–2019, 2012.
- [Ha09] Hassanzadeh, O.; Chiang, F.; Lee, H. C.; Miller, R. J.: Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1):1282–1293, 2009.
- [HM09] Hassanzadeh, O.; Miller, R. J.: Creating probabilistic databases from duplicated data. *The VLDB Journal*, 18(5):1141, 2009.
- [KFL01] Kschischang, F. R.; Frey, B. J.; Loeliger, H-A: Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.
- [KR10] Köpcke, H.; Rahm, E.: Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69(2):197–210, 2010.
- [Ku55] Kuhn, H. W.: The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.
- [Li13] Liu, X.; Yin, M.; Luo, J.; Chen, W.: An improved affinity propagation clustering algorithm for large-scale data sets. In: 2013 Ninth International Conference on Natural Computation (ICNC). *IEEE*, pp. 894–899, 2013.
- [Mu57] Munkres, J.: Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.
- [NGR16] Nentwig, M.; Groß, A.; Rahm, E.: Holistic entity clustering for linked data. In: 2016 IEEE 16th Int. Conf. on Data Mining Workshops (ICDMW). *IEEE*, pp. 194–201, 2016.
- [OSR19] Obraczka, D.; Saeedi, A.; Rahm, E.: Knowledge Graph Completion with FAMER. In: *Proc. KDD workshop Data Integration for Knowledge Graphs (DI2KG)*. 2019.
- [Pa19] Papadakis, G.; Skoutas, D.; Thanos, E.; Palpanas, T.: A survey of blocking and filtering techniques for entity resolution. *CoRR*, abs/1905.06167, 2019.
- [Ra16] Rahm, E.: The case for holistic data integration. In: *East European Conference on Advances in Databases and Information Systems*. Springer, pp. 11–27, 2016.
- [Sa18] Saeedi, A.; Nentwig, M.; Peukert, E.; Rahm, E.: Scalable matching and clustering of entities with FAMER. *Complex Systems Informatics and Modeling Quarterly*, pp. 61–83, 2018.

- [SPR17] Saeedi, A.; Peukert, E.; Rahm, E.: Comparative evaluation of distributed clustering schemes for multi-source entity resolution. In: European Conference on Advances in Databases and Information Systems. Springer, pp. 278–293, 2017.
- [SPR18] Saeedi, A.; Peukert, E.; Rahm, E.: Using link features for entity clustering in knowledge graphs. In: European Semantic Web Conference. Springer, pp. 576–592, 2018.
- [VD00] Van Dongen, S. M.: Graph Clustering by Flow Simulation. PhD thesis, University of Utrecht, 2000.
- [WB09] Wijaya, D. T.; Bressan, S.: Ricochet: A family of unconstrained algorithms for graph clustering. In: International Conference on Database Systems for Advanced Applications. Springer, pp. 153–167, 2009.
- [Zh10] Zhang, X.: Contributions to Large Scale Data Clustering and Streaming with Affinity Propagation. Application to Autonomic Grids. PARIS: University PARIS-SUD, 2010.

Appendix: Derivation of Equation (9)

In MSCD-AP, θ is a message from a factor node to a variable node and therefore is derived from Equation (11) of the max-sum algorithm, following [KFL01] and [Gi12].

$$\mu_{f \rightarrow x}(x) = \max_{n(f) \setminus \{x\}} \left(\ln f(x, y_1, \dots, y_m) + \sum_{y_i \in n(f) \setminus \{x\}} \mu_{y_i \rightarrow f}(y_i) \right) \quad (11)$$

The binary variable b_{ij} either obtains value 1 or 0. Firstly, we investigate both cases by considering *all possible configurations* of all neighboring variable nodes $b_{kj} (k \neq i)$ of t_{Qj} and then according to Equation (12) [Gi12], we combine them in order to get a scalar value for the θ message.

$$\mu_{ij} = \mu_{ij}(1) - \mu_{ij}(0) \quad (12)$$

For $b_{ij} = 1$: Equation (13) shows θ for the case that i chooses j as its exemplar. All neighbors of t_{Qj} are from the same clean source Q . Let q be the number of entities in Q . All incoming messages $\mu_{b_{kj} \rightarrow t_{Qj}}(b_{kj})$ of t_{Qj} are defined as $\gamma_{kj}(b_{kj})$. For not hurting the *clean source constraint*, no other datapoint in Q is allowed to choose j as its exemplar. Therefore all other neighboring variable nodes $b_{kj} (k \neq i)$ of t_{Qj} are set to 0. This is the only configuration that satisfies the clean source constraint and thus the optimal one. According to Equation (2), the t_{Qj} function evaluates its maximum value of 0.

$$\begin{aligned}\theta_{ij}(1) &= \max_{b_{kj}, k \neq i} [\ln t_{Qj}(b_{1j} = 0, \dots, b_{ij} = 1, \dots, b_{qj} = 0) + \sum_{b_{kj}, k \neq i} \gamma_{kj}(b_{kj} = 0)] \\ &= \sum_{k \neq i} \gamma_{kj}(0)\end{aligned}\quad (13)$$

For $b_{ij} = 0$: There is more flexibility for finding the optimal solution if datapoint i does not choose j as its exemplar. In order to guarantee the clean source consistency, utmost one of the b_{kj} variables is allowed to be set to 1. There are q possible solutions that satisfy the clean source constraint: $q - 1$ for each b_{kj} being set to 1 and one for all b_{kj} variables being set to 0. Let the case when all b_{kj} are set to 0 be x (eq. (14)) and the case when exactly one of the b_{kj} is set to 1 be y (eq. (15)). The message for $b_{ij} = 0$ in Equation (16) is the maximum of the two cases x and y .

$$x = 0 + \sum_{k \neq i} \gamma_{kj}(0) \quad (14) \quad y = \max_{k \neq i} [0 + \gamma_{kj}(1) + \sum_{p \notin \{k, i\}} \gamma_{pj}(0)] \quad (15)$$

$$\begin{aligned}\theta_{ij}(0) &= \max_{b_{kj}, k \neq i} [\ln t_{Qj}(b_{1j}, \dots, b_{ij} = 0, \dots, b_{qj}) + \sum_{b_{kj}, k \neq i} \gamma_{kj}(b_{kj})] \\ &= \max(x, y)\end{aligned}\quad (16)$$

$\theta_{ij}(1)$ and $\theta_{ij}(0)$ combined: In Equation (17) - 23, we bring both formulas for the cases $b_{ij} = 0$ and $b_{ij} = 1$ together. According to Equation (12), the scalar message is the difference of the message values for the two settings of the binary variable.

Equation (20) is transformed to Equation (21) by the transformation $a - \max(b_0, b_1, \dots, b_n) = -\max(b_0 - a, b_1 - a, \dots, b_n - a)$. Subtracting the two sums in Equation (20), only $-\gamma_{kj}(0)$ is left (eq. (22)) and then Equation (22) is transformed to Equation (23), according to Equation (12).

$$\theta_{ij} = \theta_{ij}(1) - \theta_{ij}(0) \quad (17)$$

$$= x - \max(x, y) \quad (18)$$

$$= \min(0, x - y) \quad (19)$$

$$= \min(0, \sum_{k \neq i} \gamma_{kj}(0) - \max_{k \neq i} [\gamma_{kj}(1) + \sum_{p \notin \{k, i\}} \gamma_{pj}(0)]) \quad (20)$$

$$= \min(0, -\max_{k \neq i} [\gamma_{kj}(1) + \sum_{p \notin \{k, i\}} \gamma_{pj}(0) - \sum_{k \neq i} \gamma_{kj}(0)]) \quad (21)$$

$$= \min(0, -\max_{k \neq i} [\gamma_{kj}(1) - \gamma_{kj}(0)]) \quad (22)$$

$$= \min(0, -\max_{k \neq i} [\gamma_{kj}]) \quad (23)$$

Flexible data partitioning schemes for parallel merge joins in semantic web queries

Benjamin Warnke¹, Muhammad Waqas Rehan², Stefan Fischer², Sven Groppe¹



Abstract:

Semantic Web technologies are enabling large amounts of data to be preprocessed and stored on the web to be queried efficiently later. The key technology in this topic is the triple store storing all information in the form of triples (subject, predicate and object). Depending on the triple patterns used within the queries, varying graph structures can be observed in the datasets. Currently, such properties are only exploited implicitly during join optimization in the form of histograms or similar technologies. Towards a new paradigm for explicitly exploiting graph structures in the datasets, this paper proposes a new flexible partitioning scheme at runtime. To do so, we experimented with partitioning schemes, that can be selected depending on the actual data access within a given query in order to improve query performance. The experimental results show that the proposed flexible data partitioning schemes are faster, up to a factor of 12.65 in comparison to no partitioning.

Keywords: Triple store; Partitioning; Parallel Join

1 Motivation

The Semantic Web makes huge datasets [HHK19, TWS20] available that may be queried for information processing and gathering afterwards. These datasets are continuously growing in size, either by users adding more information, or by automated data sources continuously providing new data. According to a survey paper about Semantic Web query languages [Ba05], SPARQL [SH13] is the most important RDF query language.

Some of these datasets contain more than a billion triples [HHK19]. Such big datasets are often created, maintained and used by many users. Nevertheless the users of such a Semantic Web database system desire a fast response time. Both aspects alone are challenging for a database system. Together the pressure to create fast and resource friendly query execution plans is increasing even more.

¹ Universität zu Lübeck, Institute of Information Systems, {warnke,groppe}@ifis.uni-luebeck.de

² Universität zu Lübeck, Institute of Telematics, {rehan,fischer}@itm.uni-luebeck.de

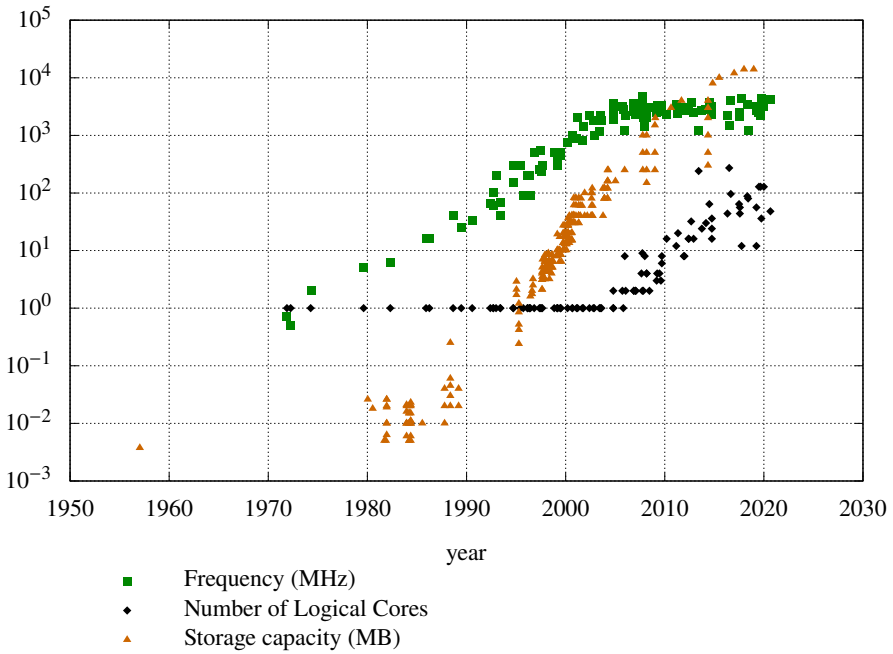


Fig. 1: CPU performance and storage size over the last 70 years. CPU-data modified from github [Ru20]. Storage-data taken from Wikipedia [Ha18]. This figure is not exhaustive and contains only data, which is available to the public.

To answer many queries on big datasets, the database system may take most out of the available hardware. The chart in Fig. 1 makes it obvious that the speed of a single processor core stagnates over the last 10 years. For further performance boosts, information technology companies have been developing CPUs with an increasing number of cores. The multi-core systems require massive parallelization for efficiently utilizing the hardware capacity and to satisfy the increasing demand for higher data processing capabilities. At the same time, the available persistent storage is increasing, too. The additional storage can be used to support massive parallelization by providing multiple variations of the original data. It opens up new challenges and opportunities for research in the context of parallel query processing.

Previous research [Ar11] has shown that various features of SPARQL such as "projection", "basic triple pattern", "join", "optional join" and "filter" are used frequently. All of these operators can be evaluated in parallel. This is advantageous because they can be evaluated even without communication between the threads, if the data is partitioned according to suitable columns. Since evaluating "projection" and "filter" in parallel is trivial, the remainder of this paper will focus on the join operator.

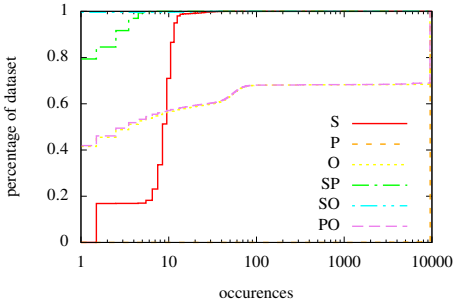
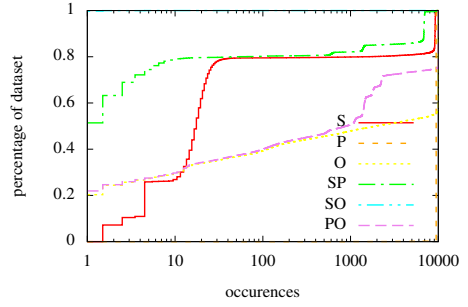
For the most part, join operators are executed using merge and hash join implementations.

Because merge joins may achieve a much higher performance than other implementations, whenever data is already sorted according to the join columns based on some previous operations or by accessing appropriate indices like B⁺-trees. Therefore we will focus on merge joins.

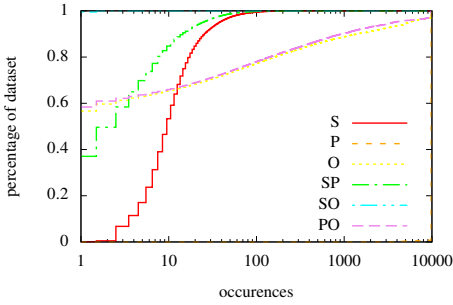
To improve data parallelism, we have analyzed the fundamental structure of data in the triple stores and the methodology of accessing it. In SPARQL, access to the stored triples is formulated in terms of triple patterns. In each triple pattern, all three components of a triple need to be specified either as a constant specifying constraints on matched triples or as a named variable for storing queried triple components.

We assume that an entirely different number of result rows can be expected based on the triple pattern being processed. On the one hand, when only the predicate is a constant, then we assume that the number of result rows are very high. In other words, a small number of predicates is required to retrieve the whole data. On the other hand, when only the predicate is a variable, then we assume that there are almost always only a few results. Similar assumptions are reasonable for other combinations of variable and constant occurrences in the triple patterns. Therefore we analyzed a synthetic dataset from the SP2B benchmark [Sc09] as well as some real world datasets, i.e. BTC2019 [HHK19], Barton[Ab07], YAGO1 [SKW07], YAGO2 [Ho13] and YAGO2s [BKS13], as shown in Fig. 2.

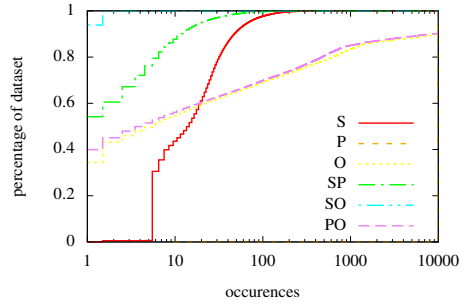
This is interesting, because data parallelism works best if there is a lot of data. That means, if only the predicate is a constant, then data partitioning may gain huge improvements. But, when only the predicate is a variable, then the improvement - if any - is small. If we consider all possible triple patterns, we come to the conclusion, that a constant in the subject position leads to few results, so that the partitioning may hardly bring any advantages. Every other triple pattern yields more results such that a benefit due to data parallelism may be likely.

(a) SP2B ($t = 2^{25}$), 33 million triples

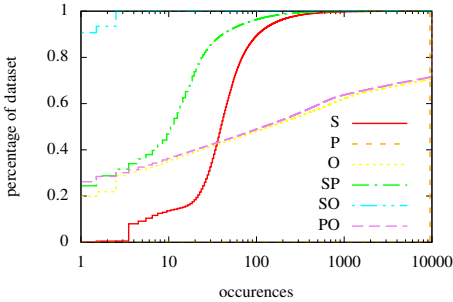
(b) Barton, 78 million triples



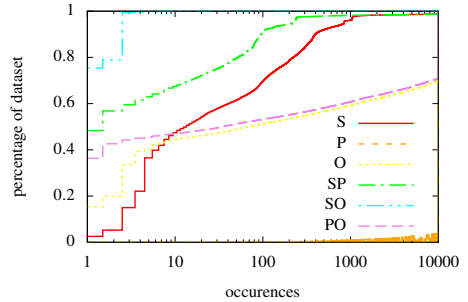
(c) YAGO1, 19 million triples



(d) YAGO2, 112 million triples



(e) YAGO2s, 171 million triples



(f) BTC2019, 256 million triples

Fig. 2: The figure shows the cumulative distribution function $f(X < x)$. The X axis shows the number of triples which share the same value at the columns specified by legend entry. The Y axis represents the percentage of the triples in the whole triple store which share their value with at most X triples. The names of the graphs consist of the constant values of the triple pattern. For example the graph P shows the relation for triple patterns of type $?s < p > ?o$, where the predicate is a constant.

The main contributions of this article are:

- Introducing a flexible partitioning scheme of the data, which allows to select the number of partitions at runtime. Therefore we store multiple different partitioning schemes at runtime.
- Analyzing the number of partitions yielding the best performance for parallel merge joins depending on the number of triple patterns and the amount of stored data.
- We propose a function to predict the optimal number of partitions depending on the data in the store and the query.
- Comparing exhaustively different parallelization strategies.
- Procuring a performance improvement up to a factor of 12.65 in comparison to no partitioning.

2 Related work

There are several ways for improving the performance of joins in the context of triple store access. The most important among them are discussed below.

2.1 Additional indices

The indices RDF3X [NW08, NW10] and Hexastore [WKB08] are often used in triple store implementations. Both index variants use a dictionary for mapping the actual values to internal numeric ids.

RDF3X [NW08, NW10] uses 6 indices, which consist of all possible collation orders of S, P and O, which are SPO, SOP, PSO, POS, OSP and OPS. The SPO index is the short form for the collation order, where the triples are first ordered by their subject, then by the predicate and finally by the object. The other collation orders only differ in which triple component is ordered first, second and last. Due to performance considerations, the ordering is applied based on the integer ids instead of the values. Additionally the ordering allows to use very efficient compression. The indices themselves are stored as B⁺-trees. Because the data is ordered, each triple pattern in the query can be translated to a range scan in the B⁺-tree.

Hexastore [WKB08] uses 6 indices with all the collation orders. Instead of a B⁺-tree, Hexastore [WKB08] uses a multi layer linking structure. Taking the SPO-index as an example, each subject points to a list of predicates, which in turn points to a list of objects. Such a list of objects can be shared with the PSO index, because they are exactly the same. Each of these lists is ordered. After the first triple is found, both indices (RDF3X and

Hexastore) allow a simple iteration over an ordered list. Both triple stores support aggressive usage of the merge join on already sorted data retrieved from its indices [NW10, WKB08].

In contrast to the above indices, which are both indexing triples, another contribution [NC20] proposes to index sub-graphs. The Triag-index [NC20] searches for triangular patterns in the graph, and stores them in a separate index. It allows a query optimizer to replace multiple consecutive joins by a range scan in their index structure. Especially in the context of ontologies it may yield high performance.

2.2 Parallel SPARQL processing

In this category, one approach [BK20] partitions the triples vertically, so that all predicates are stored in separate virtual tables. Within these tables the subjects and objects are stored in the form of independent arrays. The connection between these tables and arrays is established by vectors of pointers. The authors signify a higher storage efficiency of their approach - especially in a distributed context where it is important to find the node containing the required data.

Merge joins have a huge performance advantage over hash joins. Therefore, another approach [AKN12] partitions and orders the data on demand, so that massively parallel merge joins can be used everywhere. Even if this paper [AKN12] is about database systems in general, its results apply to Semantic Web database systems as well. Due to the sort operations at runtime, this approach requires a lot of available memory. Additionally this sorting step requires more computation time, compared to our approach, which can directly read ordered and partitioned data from the triple store.

In another paper [GG11], partitioning threads are used to horizontally partition the input data of the join operator. In this way, any intermediate result can be partitioned into any number of partitions at runtime, but at expense of the runtime overhead for partitioning. If the input for merge joins comes directly from the triple store, then partitions can be directly accessed based on the ranges in B+-trees. As a range is determined according to the histogram of the corresponding triple pattern, the partition sizes of the triple pattern to be joined may be unbalanced. This article additionally evaluates the performance gain using the pipeline parallelism. The main disadvantages of operator-based parallelism are the queues between the operators, which on the one hand require storage space and on the other hand entail thread safety by locking.

Our approach is novel in this context because it avoids the usage of partitioning threads and queues completely by employing multiple materialized balanced partitions which are flexibly chosen at query optimization time.

2.3 Distributed SPARQL processing

Our contribution focuses on local database systems. Nevertheless there exist several strategies in the distributed context, which provide different approaches to partition triple data. To the best of our knowledge, there is no distributed implementation with the same functionality as proposed in our contribution.

On top of the parallel SPARQL processing, which parallelizes on a single node, data can be distributed to multiple nodes for achieving a higher degree of parallelization. A prerequisite is obviously the presence of several nodes, as well as a fast connection between them for maintaining high performance. Similar to node local partitioning, the triples need to be assigned to a node. Typically such algorithms assign a node to each triple using a deterministic algorithm. The key task of all these algorithms is to yield a uniform distribution among all nodes.

In another paper [Ha16], the triples are distributed by a hash function on the subject. It may allow to process many joins locally on the corresponding nodes, which may reduce the network communication cost. The independent joins are similar to partitioning in a node local context. If a join can not be evaluated locally on a node, then hash joins may be heavily used for exploiting the advantage of the hash based distribution.

There are also approaches [Ha07], which allow to query from multiple data sources simultaneously. Therefore, they attach a context - the original data source - to each triple such that their database system effectively stores quads. Similar to other hash based approaches [Ha16], the hash function only uses one component of the triple for its partitioning. Hash based distribution can also be used for map-reduce based database systems [Pa13]. All of the above distribution strategies use only one component of a triple for assigning a node.

However, the following strategies use all components of a triple to calculate the assigned node. In the approach [JSL20], the connectivity between triples is calculated, and close connected triples - called "molecules" - are assigned to the same node. Apart from the huge overhead during initialization, it may allow to calculate many joins independently and locally at a node. Another approach [Ze13] uses a completely different encoding. Similar to a previous strategy [JSL20], data is distributed such that related data is stored close to each other. In this case, the data is stored in the form of adjacency-lists. The advantage is, that each node knows, which other nodes have related data. Therefore, distributed joins can explicitly access the relevant nodes, which reduces the communication overhead.

The key idea of all data distribution algorithms is to reduce or even remove the communication overhead as much as possible. As long as the data can fit in the memory of a single node, evaluating local queries is often much faster. In the remainder of this document, only node-local improvements in query evaluation are considered.

3 Our approach - flexible data partitioning

To use data parallelism, the data must be distributed over an arbitrary number of partitions. The key problem is to optimize the number of partitions. If we use too many partitions, then the overhead is larger than the benefit. If we use too few partitions, then both a fair data distribution and resource utilization (in terms of CPU core usage) is not possible.

If only merge joins are used, then the actual computation is so fast that the overhead caused by live partitioning can not be effectively compensated [GG11]. Hence, we propose to materialize different partitions already in the indices as parallel inputs to our merge join threads without introducing any computational overhead.

Fig. 2 shows that each triple pattern yields drastically different numbers of triples. Therefore, it is not possible to pick just a single number and use it for establishing the number of partitions. As a solution, we propose to use several different partitioning schemes - for each index - at the same time. This allows very flexible data storage depending on the expected data properties. Additionally, we are able to choose the used number of partitions during query optimization time. It enables a much more fine grained control over the effective parallelism. Fig. 3 shows a structural example of our proposed triple store implementation. It is important that each partitioning scheme can choose both: a different hash function and another number of partitions. We want to explicitly store partitions according to these different partitioning schemes to avoid the partitioning overhead at query runtime.

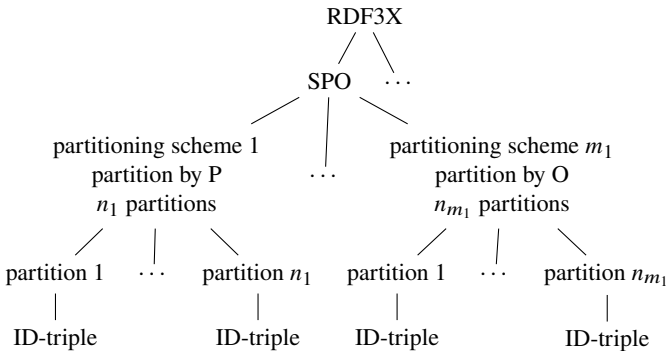


Fig. 3: Structure of database system implementation

Each partitioning scheme occupies persistent storage space. The more schemes are defined, the more memory is required. Therefore, we propose to store only a few of the most effective schemes. Unfortunately, it may introduce another type of problem requiring different numbers of partitions to join with each other.

To enable efficient implementations, we restrict ourselves to the numbers of partitions with a power of two. The efficiency comes from the properties of the modulo operator. If the number of partitions is halved, then exactly two partitions need to merge to a single one -

without touching any other partition. It allows to use much less locking, because less threads share common locks, and therefore increases the speed. In Sect. 4 we evaluate where to use which partitioning scheme.

4 Evaluation

To gain insight into how the number or presence of an additional partitioning layer affects the database system performance, there are many performance aspects to consider. In this paper, we focus on parallel main-memory query evaluation of merge joins.

4.1 Experimental setup

We use two different Benchmark Systems to verify the validity of our findings independently of the hardware architecture. That enables us to compare an server CPU with a recently introduced desktop CPU of the current generation.

The first machine (M1) is a dual socket machine using Intel Xeon E5-2620 v3 CPUs with a clock rate of 2.4GHz. In our experiments, hyper threading is enabled such that there are 24 hardware supported threads. Each socket is assigned with 16GB memory, such that a total of 32GB memory is available. The database systems either use gcc 5.4 or Java 1.8.265 in the server-edition.

The other machine (M2) is a single socket machine using an Intel i9-10900K CPU with a clock rate of 4.9 GHz. On this machine, hyper threading is enabled as well, such that there are 20 hardware supported threads. This machine has 64GB of RAM installed. It uses Java 14.0.2 in the server-edition.

4.2 Implementation details

Our database system LUPOSDATE3000³ is a rewrite of LUPOSDATE [Gr11]. The old LUPOSDATE [Gr11] is implemented in Java. The new LUPOSDATE3000 database system is implemented in Kotlin programming language in order to support different targets like the JVM, JavaScript and native binaries for desktop, server, web and mobile environments. Currently, Kotlin-JVM-target is the fastest, therefore all benchmarks are evaluated using Java runtime. LUPOSDATE3000 uses a dictionary to map all values to integer IDs. These IDs are then stored in an index similar to RDF3X using all 6 collation orders. During query evaluation, LUPOSDATE3000 uses both column and row iterators, preferring the column iterators where applicable. In our experiments, the hash function used for partitioning only

³ <https://github.com/luposdate3000/luposdate3000.git>

performs the modulo operator to the integer IDs in the store. We yield uniform partition sizes with these hash functions.

We choose Apache Jena⁴, blaze-graph⁵ and virtuoso⁶ as competitive database systems, because they are the most used open source RDF database systems according to an RDF database ranking [DB20].

Jena is an RDF store written in Java. Since we use Kotlin-JVM-target, it allows to compare two different database systems using the same Java runtime environment. The triples are stored in B⁺-trees.

Blaze-graph is written in Java. The indexes are stored in B⁺-trees which are influenced by Google's BigTable system.

Virtuoso is written in c++. In our tests, we only use the RDF interface of virtuoso. It allows the comparison to a compiled database system, and verifies our expectation that garbage collected languages are not inherently slow.

4.3 Datasets and queries

In order to facilitate clear indications about how exactly partitioning influences the execution times, we use simple queries as shown in Fig. 4. The query Q1 is used as a template for the benchmarks which use up to 16 consecutive merge joins. Q2 is the only query, which enforces a hash join, all other queries can be evaluated using only merge joins.

```
PREFIX b: <http://benchmark.com/>
SELECT *
WHERE {
  ?s b:p0 ?o0 .
  ?s b:p1 ?o1 .
  ?s b:p2 ?o2 .
}
```

(a) Query Q1

```
PREFIX b: <http://benchmark.com/>
SELECT *
WHERE {
  ?s b:p0 ?o0 .
  ?s b:p1 ?o1 .
  ?o1 b:p2 ?o2 .
}
```

(b) Query Q2

Fig. 4: SPARQL queries used for the benchmarks.

We create synthetic data such that it matches our requirements of selectivity and output size. Because our proposed changes are not related to join order optimization, we do not want any side effects of the applied optimizer. Therefore our generated triple structure is the same for every triple pattern in our query.

⁴ Version 3.14.0

⁵ Version 2.1.6

⁶ [git://github.com/openlink/virtuoso-opensource.git](https://github.com/openlink/virtuoso-opensource.git), Revision 840b468fc400a254eab0eb20f1afde6ca3c2220d

We label all our graphs with result rows instead of the usually used number of triples. In this way, we can enforce a uniform workload across all the used threads. Furthermore, a low selectivity combined with a low number of input triples would yield too few result rows without notice. A few results would not be uniformly distributed to the threads, which in turn decreases the benefit achieved from multiple threads.

4.4 Query optimizer

Due to the new partitioning scheme, there are lots of possibilities how to execute a simple query. The query Q1 shown in Fig. 4a can be evaluated using two merge joins. We want to compare the performance of 1, 2, 4, 8 and 16 partitions for each of the join operators as well as for each of the triple store iterators. We remove the options, where a merge join requires to change the partitioning of both of its inputs during the runtime, therefore we get $2^2 \cdot 4^3 = 256$ different operator graphs.

Using the same number of partitions everywhere in the operator graphs yields the best results. We have verified it using the computer configurations of M1 and M2. Additionally in our experiments, we have considered different synthetic datasets, too, with either uniform or non uniform data distributions. For the non uniform synthetic datasets, we increased the differences such that one triple pattern yields up to 128 times more input than the others. This has changed the total query evaluation time, but not the optimal partitioning ranking in the operator graph.

Depending on where exactly the number of partitions changes in the operator graph, the evaluation speed is not that much lower compared to operator graphs using only one partition count. It means that our approach does not need to assign the same number of partitions to every collation order.

The query Q2 shown in Fig. 4b joins on two different variables, which means, that we can not use two merge joins. It yields to some more options for the optimizer especially which number of partitions on which variable are used for the second join. One option is, to merge the partitions after the first join, and then change the partitioning of that result just in time for the second join. The other option is to pass through the partitioning as it is, which means, that the second join is not partitioned by its join variable. In this case the second join must read in the whole not partitioned or united input from the other side, to still produce valid results. Our measurements show, that in this case it is the fastest to use the second option, which is passing through data which is not partitioned by a join variable.

4.5 Benchmarks

This section is divided into two parts. The first part deals with the evaluation of a micro-benchmark with an explicit focus on the performance of the merge join operator. Therefore, several database system functions are disabled or bypassed. The changes made are:

- The operator graphs, including the partitions to use, are hard coded, in order to investigate their effects on query performance.
- The benchmark code is included in the LUPOSDATE3000 binary to avoid HTTP-interface overhead.
- The result is only calculated as a row of integer IDs. The required dictionary lookups for converting those IDs to Strings are not included in the benchmarks.

This benchmark is intended to show the effect on query evaluation, in case both data structure and data size are changed. Since, it is not easy to apply the above changes to other existing database systems, this part is only evaluated within LUPOSDATE3000.

In the second part, the same queries are evaluated on multiple database system implementations. Here in this part, all database system features are enabled, and the HTTP-SPARQL endpoints are used.

4.5.1 Micro benchmark

To focus on the performance effects of partitioning during query processing of join operator chains, a synthetic micro-benchmark is used so that the input data has the desired input patterns. During the tests, the following properties have an impact on the query runtime:

- Number of input rows and number of result rows: Larger numbers of rows yield higher speedup because the sequential query initialization phase needs less time compared to the total computation time.
- Selectivity of the joins: When more rows are filtered away, the next join operator has less work to do, resulting in faster query processing.
- Number of CPU-cores: Due to the in-memory benchmark-setup, all experiments are both memory and CPU bounded. As long as the data is evenly distributed, it does not make any sense to employ more partitions than CPU cores.
- Number of partitions: Currently LUPOSDATE3000 parallelizes its query evaluation based on partitions only. As a result, maximum speedup corresponds to the number of partitions. Depending on the other parameters the maximum speedup may not be reached.

- Number of consecutive joins: More consecutive joins on the same join columns increase the throughput because there is no need to serialize or cache intermediate results.

Since we are interested in comparing the effects of different selectivities within the join operators, we have generated multiple synthetic datasets. For generation of the datasets we have used the following strategies:

For selectivities lower than 1, we choose a fixed n which specifies how many triples to skip after we find a triple participating in a join. Then, we repeat this procedure during data generation for every basic triple pattern. It yields a selectivity of $\frac{1}{1+n}$ in each join operator. In the following we restrict n to be a power of two.

Additionally, we have generated datasets where data volume is increasing within the join operators. To create such datasets, we emit blocks of m triples, which are then joined with each other. In the following we choose m to be a power of two. In the experiments we call this "selectivity", too, because it still specifies the factor by which the number of rows changes within the joins.

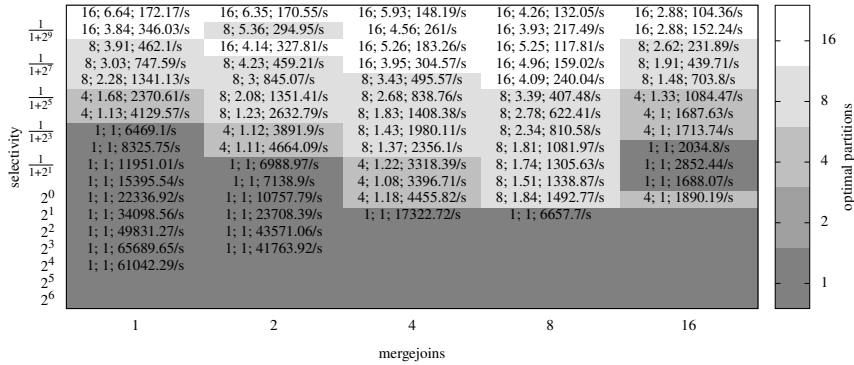
To generate our queries, we use the SPARQL template as shown in Fig. 4a. Afterwards, we change the number of triple patterns according to the desired number of joins.

In Fig. 5 we can see, that all of the three properties (output-rows, selectivity and number of joins) affect the optimal number of partitions for evaluating a query.

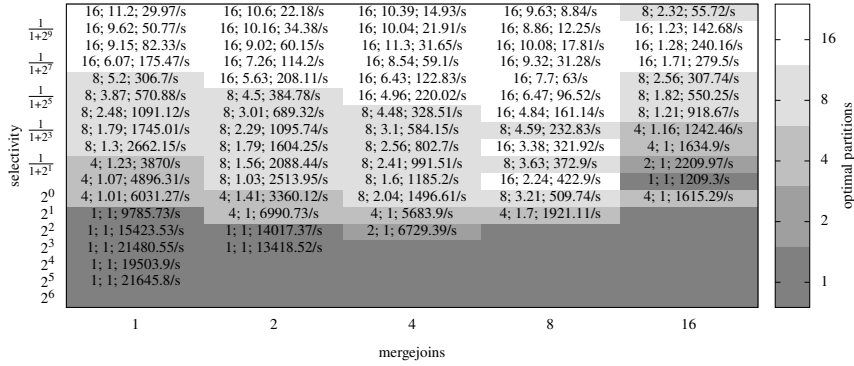
In the bottom right of each figure, there are missing experiments, because we can not create the target number of output rows. This is due to a massive increase of rows within the join operator chain, which is higher than the targeted output row count. Low optimal partition numbers in the bottom left part of each figure are caused by the same reason. Due to the very small number of triples in the store, it does not make sense to apply partitioning.

The optimal number of partitions is proportional to the theoretical workload, which we had expected. The more triples are stripped away due to a low selectivity, the more triples must be available in the store in the first place. The same holds for the number of merge joins. The more distinct triple patterns we want to join, the more input triples need to be defined. The last property, increasing the number of output rows obviously requires an increased number of input rows too. Vice versa, if we would have fixed the number of input rows, we would yield a similar result. In that case, the optimal number of partitions would be proportional to the number of output rows.

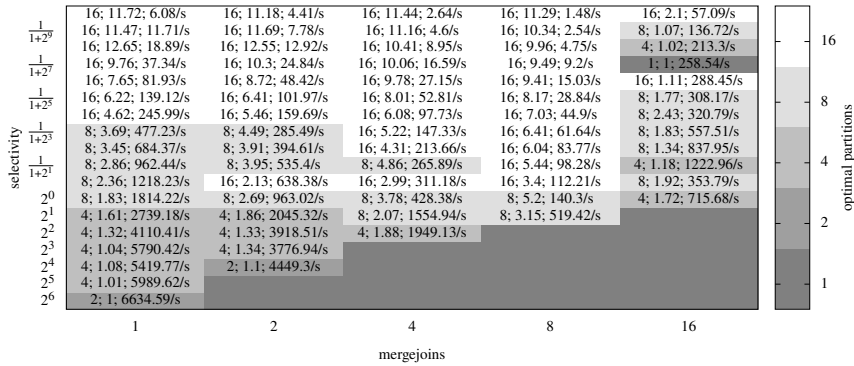
We used the results of this benchmark to predict the fastest partitioning within the query optimizer. As a base function we choose the polynomial $a \cdot x + b \cdot y + c \cdot z + d \cdot x^2 + e \cdot y^2 + f \cdot z^2 + g$ with "x" as the number of result rows, "y" as the number of joins and "z" as the expected selectivity. We choose this function, because it promises good results for predicting the fastest partitioning and the function can be evaluated very fast during the query optimization



(a) 512 result rows



(b) 2048 result rows



(c) 8192 result rows

Fig. 5: Optimal number of partitions depending on the number and the selectivity of merge joins. The labels follow the form "a;b;c", where "a" is the optimal number of partitions, "b" the speedup compared to no partitions and "c" the queries per second when no partitions are used. These experiments are run on computer configuration M2, because the additional RAM allows more experiments to be evaluated. Evaluating the queries on M1 achieve similar results.

phase. For our machine M2 we calculated the constants "a" to "g" such that we yield the complete function as seen in Fig. 6. We use the helper function $h(z)$ to convert the selectivity values to a similar range of numbers as all of the other variables. In a final step, we round the value to a discrete number of partitions. The prediction function $p(x, y, z)$ calculates most numbers of partitions optimally. Nevertheless we calculated the mean squared error between Fig. 6 and Fig. 5 to be 4.3030, which is quite small in comparison the huge number of known value pairs to fit.

$$\begin{aligned}
 h(z) &= -\log_2(z) \\
 f(x, y, z) &= 0.0025 \cdot x + 1.4827 \cdot y + 1.1277 \cdot h(z) + 0.0906 \cdot y^2 + 0.0279 \cdot h(z)^2 - 3.3696 \\
 p(x, y, z) &= 2^{\lfloor \log_2(f(x, y, z)) \rfloor}
 \end{aligned}$$

Fig. 6: Prediction function for the number of partitions to use

Even if the above benchmark is evaluated on various synthetic datasets with different queries, we see the same effects in real world data, too. Every time a query uses a different constant (for example as a predicate), a completely different subgraph is accessed. Each subgraph in a real world dataset may contain a different number of triples as we have analyzed in Fig. 2. When we pick two different subgraphs, and join them with each other, we get very different selectivities within the join operators as well as different numbers of output rows, by only changing the query. This approves our assumption, that by just changing the query, the optimal partitioning scheme changes.

4.5.2 Macro benchmarks

This section presents a macro benchmark for comparing the overall speed of our LU-POSDATE3000 database system in a macro benchmark to other database systems in this section.

The database system LUPOSDATE was configured to use either its in-memory storage or a disk based RDF3X storage layout. Consequently, the results from both in-memory storage and disk based RDF3X storage layout are presented because the internal implementation of the triple stores is completely independent. All other database systems are using their default parameters as suggested by their documentation.

The performance measurements of the blaze-graph database system are suffering from very large measurement inaccuracies. All other database systems have a very low variation in the required time for the same query. To counter these inaccuracies, we repeated all experiments 10 times - and use the average measurements in our graphs.

For the experimental comparison, we have executed 10 merge joins on several different

datasets. These datasets are generated in a manner to yield the target selectivity within each join operator. Fig. 7 shows the results of this comparison.

We have chosen to fix two different result sizes, and plot the graphs over a changing join selectivity. Both figures highlight different effects. The experiments of the Jena database system as well as the in-memory LUPOSDATE in the 128 result rows setting are dominated by the effect, that a fixed output size with a decreasing selectivity requires an increasing amount of input data.

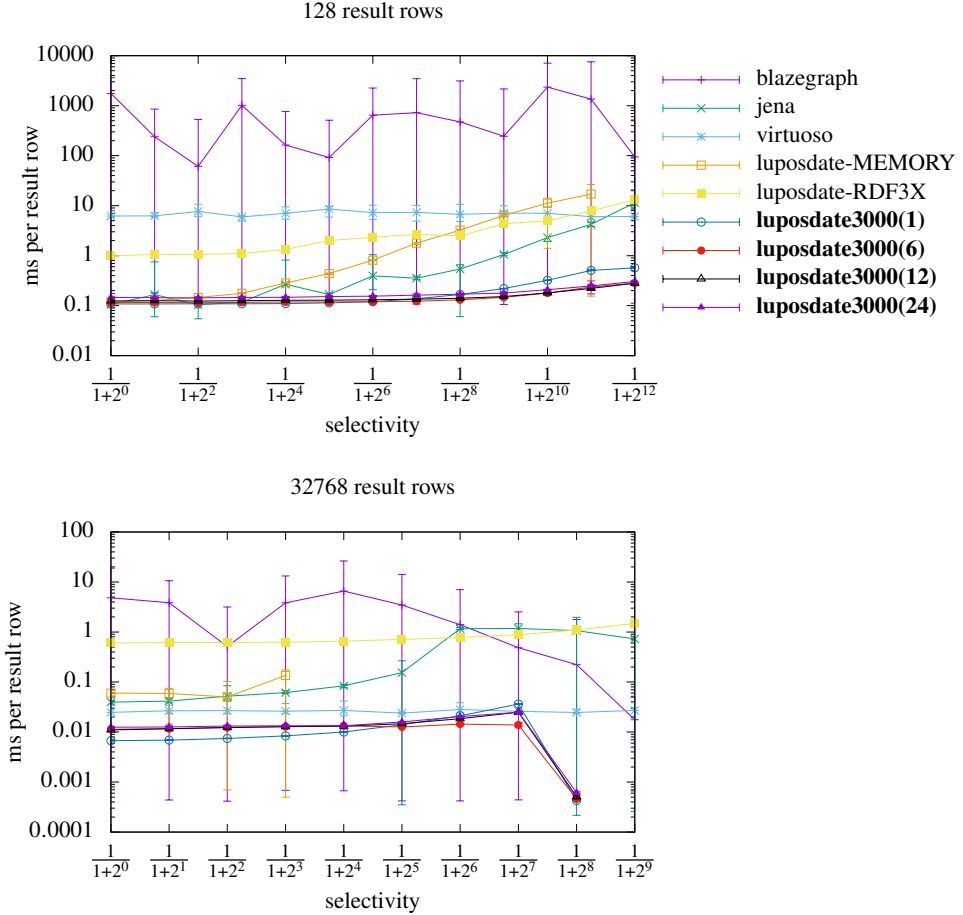


Fig. 7: Performance of Q1 with 10 merge joins on different database systems. These benchmarks are evaluated on M1 only, because we have multiple instances available, to perform more experiments. The numbers in the brackets, for example LUPOSDATE3000(6), show the number of used partitions.

Starting with a selectivity of $\frac{1}{1+2^9}$ the sequential performance of LUPOSDATE3000 decreases, while the time requirement of the partitioned evaluation remains the same. Although for the macro benchmarks the execution times for LUPOSDATE3000 include overhead for the endpoint communication and materialization of the string representations of the values, we still achieve speedups up to a factor of 1.81 compared to no partitioning, which shows that the correct number of partitions is significant for query evaluation.

All other configurations i.e. virtuoso, LUPOSDATE using RDF3x and partitioned LUPOSDATE3000 require the same evaluation time completely independent of the selectivity of the join operators, because very small data causes the database system to spend most of its time in static initialization.

When 32768 result rows are used, Fig. 7 shows a completely different performance characteristic, even if the only difference in the benchmark setup is the higher number of result rows. virtuoso as well as LUPOSDATE with RDF3X are still unaffected by changing the selectivity. This indicates that both of these database systems are limited by their capability to output their finished results. Contrary to before the required time per row decreases with decreasing selectivity for all LUPOSDATE3000 configurations as well as blaze-graph. This effect is based on the fact, that the static initialization time is getting smaller compared to the total evaluation time. Therefore, the overall speed per result row increases. The in memory LUPOSDATE variant suffers from bad memory management because e.g. no dictionary is used to map string representations to integer identifiers. Dictionaries are decreasing the memory footprint in all other database systems such that out-of-memory-errors are avoided during the triple load phase.

Even if the micro benchmark shows, that for huge numbers of result rows more partitions are better, the macro benchmark in the 32768 result rows setting is faster, if less partitions are used. At the same time both benchmarks show, that the sequential execution is slower. We believe, that this is caused by the different benchmark setup. Especially the sequential text output requires synchronization between the threads, which is not needed in the micro benchmark.

5 Summary and future work

In this paper we have investigated the factors which impact the performance of parallel SPARQL query processing. We have focused on the performance of data parallelism. According to our experiments, the performance depends on the amount of data in the store, the available hardware, and the structure of the query to process. In order to avoid overhead introduced by additional partitioning phases, we propose that the triple store materializes multiple independent partitioning schemes and choose the best among them on the fly. We present an experimental analysis and a concept of how a database system may optimize its query processing in this multi partitioning scheme context.

In the future we will implement and evaluate this partitioning strategy in a distributed database context. Due to our choice of Kotlin as our implementation-language, we will be able to run our database implementation directly on various operating systems. We plan to use these possibilities to evaluate our approach in a multi-operating-system environment like the Internet of Things with extremely heterogeneous hardware components. We expect that our approach will have huge advantages in those environments.

Acknowledgements

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 422053062

Funded by



Deutsche
Forschungsgemeinschaft
German Research Foundation

Bibliography

- [Ab07] Abadi, Daniel J.; Marcus, Adam; Madden, Samuel R.; Hollenbach, Kate: Using The Barton Libraries Dataset As An RDF Benchmark. Technical Report MIT-CSAIL-TR-2007-036, MIT, 2007.
- [AKN12] Albutiu, Martina-Cezara; Kemper, Alfons; Neumann, Thomas: Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. Proc. VLDB Endow., 5(10):1064–1075, June 2012.
- [Ar11] Arias, Mario; Fernández, Javier D; Martínez-Prieto, Miguel A; de la Fuente, Pablo: An empirical study of real-world SPARQL queries. arXiv preprint arXiv:1103.5043, 2011.
- [Ba05] Bailey, James; Bry, François; Furche, Tim; Schaffert, Sebastian: Web and Semantic Web Query Languages: A Survey. In: Reasoning Web, pp. 35–133. Springer Berlin Heidelberg, 2005.
- [BK20] Bilidas, Dimitris; Koubarakis, Manolis: In-memory parallelization of join queries over large ontological hierarchies. Distributed and Parallel Databases, June 2020.
- [BKS13] Biega, Joanna; Kuzey, Erdal; Suchanek, Fabian M: Inside YAGO2s: a transparent information extraction architecture. In: Proceedings of the 22nd International Conference on World Wide Web. pp. 325–328, 2013.
- [DB20] DB-Engines Ranking of RDF Stores. <https://db-engines.com/en/ranking/rdf+store>, 2020. Accessed: 2020-09-16.
- [GG11] Groppe, Jinghua; Groppe, Sven: Parallelizing join computations of SPARQL queries for large semantic web databases. In: Proceedings of the 2011 ACM Symposium on Applied Computing. pp. 1681–1686, 2011.
- [Gr11] Groppe, Sven: Data Management and Query Processing in Semantic Web Databases. Springer, 2011.

- [Ha07] Harth, Andreas; Umbrich, Jürgen; Hogan, Aidan; Decker, Stefan: YARS2: A federated repository for querying graph structured data from the web. In: *The Semantic Web*, pp. 211–224. Springer, 2007.
- [Ha16] Harbi, Razen; Abdelaziz, Ibrahim; Kalnis, Panos; Mamoulis, Nikos; Ebrahim, Yasser; Sahli, Majed: Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal*, 25(3):355–380, 2016.
- [Ha18] Hankwang: , Hard drive capacity over time. https://de.wikipedia.org/wiki/Datei:Hard_drive_capacity_over_time.svg, 12 2018. Accessed: 2020-09-16.
- [HHK19] Herrera, José-Miguel; Hogan, Aidan; Käfer, Tobias: BTC-2019: The 2019 Billion Triple Challenge Dataset. In: *International Semantic Web Conference*, Auckland, New Zealand. Springer, pp. 163–180, 2019.
- [Ho13] Hoffart, Johannes; Suchanek, Fabian M; Berberich, Klaus; Weikum, Gerhard: YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence*, 194:28–61, 2013.
- [JSL20] Janke, Daniel; Staab, Steffen; Leinberger, Martin: Data placement strategies that speed-up distributed graph query processing. In: *Proceedings of The International Workshop on Semantic Big Data*. pp. 1–6, 2020.
- [NC20] Naacke, Hubert; Curé, Olivier: On Distributed SPARQL Query Processing Using Triangles of RDF Triples. *Open Journal of Semantic Web (OJSW)*, 7(1):17–32, 2020.
- [NW08] Neumann, Thomas; Weikum, Gerhard: RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.
- [NW10] Neumann, Thomas; Weikum, Gerhard: The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.
- [Pa13] Papailiou, Nikolaos; Konstantinou, Ioannis; Tsoumakos, Dimitrios; Karras, Panagiotis; Koziris, Nectarios: H 2 RDF+: High-performance distributed joins over large-scale RDF graphs. In: *2013 IEEE International Conference on Big Data*. IEEE, pp. 255–263, 2013.
- [Ru20] Rupp, Karl: , microprocessor-trend-data. <https://github.com/karlrupp/microprocessor-trend-data>, 7 2020.
- [Sc09] Schmidt, Michael; Hornung, Thomas; Lausen, Georg; Pinkel, Christoph: SP²Bench: a SPARQL performance benchmark. In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE, pp. 222–233, 2009.
- [SH13] Seaborne, Andy; Harris, Steven: SPARQL 1.1 Query Language. W3C recommendation, W3C, March 2013. <https://www.w3.org/TR/2013/REC-sparql11-query-20130321/>.
- [SKW07] Suchanek, Fabian M; Kasneci, Gjergji; Weikum, Gerhard: Yago: a core of semantic knowledge. In: *Proceedings of the 16th international conference on World Wide Web*. pp. 697–706, 2007.
- [TWS20] Tanon, Thomas Pellissier; Weikum, Gerhard; Suchanek, Fabian: YAGO 4: A Reason-able Knowledge Base. In: *European Semantic Web Conference*. Springer, pp. 583–596, 2020.

- [WKB08] Weiss, Cathrin; Karras, Panagiotis; Bernstein, Abraham: Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.
- [Ze13] Zeng, Kai; Yang, Jiacheng; Wang, Haixun; Shao, Bin; Wang, Zhongyuan: A distributed graph engine for web scale RDF data. *Proceedings of the VLDB Endowment*, 6(4):265–276, 2013.

Multi-Party Privacy Preserving Record Linkage in Dynamic Metric Space

Ziad Sehili¹, Florens Rohde², Martin Franke³, Erhard Rahm⁴

Abstract:

We propose and evaluate several approaches for privacy-preserving record linkage for multiple data sources. To reduce the number of comparisons for scalability we propose a new pivot-based metric space approach that dynamically adapts the selection of pivots for additional sources and growing data volume. Furthermore, we investigate so-called early and late clustering schemes that either cluster matching records per additional source or holistically for all sources. A comprehensive evaluation for different datasets confirms the high effectiveness and efficiency of the proposed methods.

Keywords: Privacy Preserving Record Linkage, Bloom filter, metric space, triangle inequality, Clustering

1 Introduction

Record linkage or entity resolution is the task of finding records in different data sources that describe the same real-world entity, e.g. product or customer. Privacy-preserving record linkage (PPRL) is a special form of record linkage for sensitive data and aims at achieving record linkage while preserving privacy. This kind of record linkage is especially important for person-related record linkage, e.g., for finding matching patient or customer records.

PPRL has received a substantial amount of research interest in the last decade [SBR09, VCV13, Va17]. Most of the proposed approaches aim at improving privacy by matching on encoded record attribute values instead of the original values for identifying attributes, such as person name, address and birth of date. These attributes are called quasi-identifiers (QIDs) as the equality or high similarity in these attributes allows one to find matching persons. Many proposed PPRL schemes also rely on a so-called trusted linkage unit to perform the matching of encoded person records thereby avoiding the need to exchange records between different data owners [VCV13]. Like normal record linkage, PPRL has to achieve a high match quality and scalability to large datasets.

¹ Leipzig University, Database Group, Germany sehili@informatik.uni-leipzig.de

² Leipzig University, Database Group, Germany rohde@informatik.uni-leipzig.de

³ Leipzig University, Database Group, Germany franke@informatik.uni-leipzig.de

⁴ Leipzig University, Database Group, Germany rahm@informatik.uni-leipzig.de

Most proposed PPRL methods focus on the special case of linking two sources only [VCV13]. However, various use cases and data analysis tasks require a PPRL for multiple (≥ 2) sources, e.g., databases from hospitals, clinical studies and census data. Multi-Party PPRL (MP-PPRL) introduces further challenges to be addressed. In particular, the number of record comparisons grows quadratically with the size and the number of sources making scalability a problem. Furthermore, a record may have matches in an arbitrary subset of the data sources not only in one data source. This asks for clustering matching records over multiple sources so that a cluster contains all matches for a specific person. This clustering should utilize that individual sources are often curated and duplicate-free so that every cluster should have at most one record for any data source.

To address these challenges we investigate several novel approaches for MP-PPRL and clustering of encoded records. Specifically, we make the following contributions:

- To reduce the number of comparisons between records we utilize a pivot-based metric space approach [SR16]. We propose an extension of the static approach with a dynamic adaptation of the pivot selection in order to deal with additional data sources and growing data volume.
- We investigate different clustering schemes for multiple parties that either cluster new data sources one after the other (early clustering) or that first determine similar record pairs over all sources before a final clustering is performed (late clustering).
- The presented approaches, PPRL in dynamic metric space and the clustering techniques, are exhaustively evaluated using synthetic and real datasets. We also compare them with baseline approaches.

After a brief discussion of related work we describe basics of PPRL (including the use of Bloom filters to encode quasi-identifiers) and metric space. Section 4 describes the dynamic pivoting approach for multi-party PPRL. The early and late clustering approaches are outlined in Section 5. In Section 6 we evaluate the proposed approaches before we conclude.

2 Related Work

PPRL has been applied in several real health-related use cases, e.g., to compare surgical treatment received by Aboriginal and non-Aboriginal people with non-small cell lung cancer in Australia [Gi16], or to analyze long-term consequences of childhood cancer in Switzerland by linking national data from several cantonal registries [Ku11]. Several surveys [VCV13, Va17, BRF15] categorize the variety of proposed PPRL methods with respect to their challenges of privacy, quality and scalability. The privacy of the represented entities can be provided by using either secure multiparty computation (SMC) to run PPRL between the data owners without involving a linkage unit or by encoding records, e.g. within

Bloom filters [SBR09], and then sending the encoded records to a dedicated linkage unit (see Section 3). Although SMC methods for PPRL aim at higher privacy guarantees they are not applicable in many practical use cases due to their high computational complexity. The linkage quality depends on many factors like the input data, the linkage and classification methods [Ch12], and the encoding technique. While the first factors are not specific to PPRL but also apply to record linkage in general, the effect of encoding records to Bloom filters before comparing them have been studied in [SBR11, Du12]. The authors have shown that the linkage quality of Bloom filters does not drop significantly compared to the original (string) records. We will thus use Bloom filters for encoding records in this paper.

In previous PPRL papers, the scalability problem has been addressed by applying blocking or filtering techniques for Bloom filters to reduce the number of comparisons [Se15, SR16]. Furthermore, the PPRL computations at a linkage unit can be run in parallel [GI18, FSR18]. All previous blocking and filtering schemes for PPRL are static so that their effectiveness tends to decrease with increasing data volume (e.g. the number of comparisons per block mostly grows quadratically with the block size). In our previous work [SR16] we introduced the use of a pivot-based metric space approach [Ze06] for PPRL with two sources. In contrast to blocking, this approach can reduce the number of comparisons without introducing from recall reductions. The main drawback of this method is its upfront selection of a static set of reference records (pivots) making it inapplicable for dynamically growing number and size of sources. To overcome this problem we advise a new dynamic pivoting scheme. In a non-PPRL setting a related dynamic approach, Sparse Spacial Selection (SSS), has been proposed in [PB07], that dynamically selects reference records from one source depending on the spacial distribution of records in the metric space. We use this method as baseline in our evaluation.

Multi-party record linkage with a clustering of matching records without the privacy prerequisite was studied in [Sa18]. The approach uses a static blocking scheme, determines similar pairs of records from all sources (and keeps them in a similarity graph) before a final clustering is performed (late clustering). [Fr18] shows that PPRL match results for two duplicate-free sources can be improved by a post-processing to determine a 1:1 match mapping. This can be achieved by a Hungarian or Max-Both approach and we will use these methods in our early clustering schemes for multiple parties. [VCR20] investigates clustering techniques for MP-PPRL, but for static blocking instead of our dynamic metric space filtering.

3 Preliminaries

This section starts by introducing the multi-party PPRL process, then presents the Bloom filter encoding scheme that preserves privacy and similarity of records. Furthermore, we explain the use of metric space to improve scalability of the linkage process.

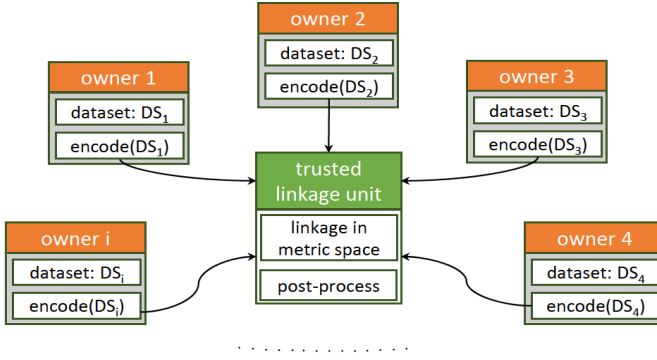


Fig. 1: Multi-Party PPRL: The data owners encode their records and send them to a trusted linkage unit (LU). The LU links the records and determines clusters of matching records

3.1 Multi-Party PPRL

A Multi-Party PPRL process that involves i data holders and a trusted linkage unit (LU) to run the linkage process is depicted in Fig. 1. We assume duplicate-free datasets DS_1, DS_2, \dots, DS_i . Quasi-identifying attributes of the records are first encoded to Bloom filters (see below) by their respective data holders to preserve the privacy of the represented persons. The encoded records are sent to the LU where a linkage process using the metric space approach is applied for improved scalability (explained in Sec. 4) and where clusters of similar records are generated. Furthermore, and due to the assumption that the sources are duplicate-free, a post-processing or cleaning step is run to ensure that each cluster contains at most one record from any source. Hence, the size of any clean cluster c is $1 \leq |c| \leq i$. Clustering strategies are discussed in Sec. 5.

3.2 Bloom Filter

The use of Bloom filter [BI70] to encode records involved in PPRL was introduced in [SBR09]. The encoding scheme of one record takes as input a set of q -grams (bi- or tri-grams) of the relevant attributes (e.g. first and last name, date of birth and address), a bit array of length l with all positions initially set to zero and a set of k independent cryptographic hash functions that return values in $[0, l - 1]$. Then each q -gram is mapped to the bit array k -times using the k hash functions by setting the corresponding positions to 1. Figure 2 shows a simple example of two similar names and their encoding to Bloom filters of length $l = 20$ using $k = 2$ hash functions.

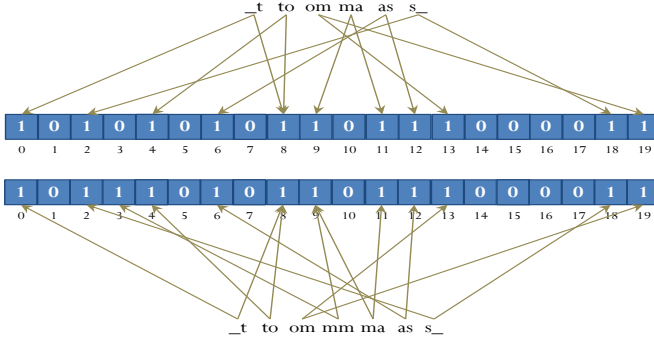


Fig. 2: Bloom filter (bit vector) encoding of two names *tomas* and *tommas*, each tokenized to bi-grams, using $k = 2$ hash functions and bit vectors of length $l = 20$

3.3 PPRL in Metric Space

A metric space $\mathcal{M}(\mathcal{U}, d)$ consists of a set of data objects \mathcal{U} and a metric d to compute the distance between these objects. The main property satisfied by the distance metric d is the *triangle inequality* of the distances: $\forall x, y, z \in \mathcal{U} : d(x, z) \leq d(x, y) + d(y, z)$. This inequality can be used to considerably reduce the search space without discarding any pair of similar records, i.e., without reducing recall [Ze06].

For a data object $q \in U$ the match candidates cannot be outside a radius $rad(q)$ in order to satisfy a maximal distance threshold. The triangle inequality allows one to avoid computing the distance between points x and q based on precomputed distances to a reference point p , also known as pivot. Hence, only objects that satisfy:

$$|d(p, q) - d(p, x)| \leq rad(q) \quad (1)$$

need to be considered as potential matches so that the distance $d(q, x)$ has to be computed to determine whether q and x match.

PPRL processes generally use a similarity function like Jaccard to compare records and a threshold t to classify pairs as similar or non-similar. Since metric spaces rely on metrics (distance function), we can use the Hamming distance, that was shown to be equivalent to the Jaccard similarity [Xi08], to search matching records. The radius $rad(q)$ that includes similar records to q can be inferred from t as [SR16]:

$$rad(q) = |q| \times \frac{1 - t}{t}.$$

where $|q|$ is the number of positions set to 1 in the Bloom filter. Records having a Hamming distance $\leq rad(q)$ are those that have a Jaccard similarity $\geq t$.

For two sources DS_1 and DS_2 , the pivot-based metric space approach for PPRL operates in two steps: index building and similarity search. Indexing source DS_1 first requires selecting a set P of m data objects that serve as pivots. One proven approach is to select these pivots from DS_1 in an iterative manner by choosing the objects having the maximum distance to each other [SR16]. Next, each object $x \in DS_1$ is assigned to its closest pivot $p_i \in P$, $elem(p_i) = \{x \in DS_1 : dist(x, p_i) < dist(x, p_j), \forall j \neq i\}$. For every object $x \in DS_1$ the precomputed distance $d(x, p_i)$ to its pivot p_i is stored as well as the radius of a pivot $rad(p_i)$, i.e., the maximal distance for any object assigned to pivot p_i .

For each (query) object $q \in DS_2$, the matching with DS_1 objects involves a similarity search with radius $rad(q)$. This search for match candidates can utilize two filter steps. First, only those pivots p_i need to be considered for which the pivot radius $rad(p_i)$ overlaps with $rad(q)$ since otherwise all of the pivot's objects are outside radius $rad(q)$ and cannot match. Secondly, for the remaining pivots p_i the number of its objects x can be reduced according to the above triangle inequality.

4 MP-PPRL in Dynamic Metric Space

We first explain our approach to dynamically increase the number of pivots for growing data volume and then explain the use of this approach for multi-party PPRL.

4.1 Dynamic Pivoting

The static pivot method with its upfront determination of pivots has two problems: 1) the number of pivots is difficult to determine and depends on the number and distribution of records. 2) The number of pivots should be adequately increased with growing number and size of sources to be indexed since more records per pivot leads to more comparisons and thus poor scalability. Furthermore, more records per pivot lead to increased pivot radii and higher overlap between the pivots (see below) which in turn can cause that more pivots need to be considered to find the matches of a query record. Hence, the potential to reduce the number of comparisons can be severely reduced when we keep a static set of pivots in the presence of strongly growing data volume, e.g. due to additional data sources.

To overcome these problems for multi-source PPRL, we devise an algorithm to adapt the pivot selection dynamically during the indexation of an additional data source. For this purpose we control the so-called pivot overlap using a parameter α . Note that an indexed record x may be in the radius of several pivots p_i ($dist(x, p_i) < rad(p_i)$) although it is assigned to only one (the closest) pivot. Similar to the idea introduced in [Tr00], the overlap between pivots can thus be determined by the average number of intersections per record.

Algorithm 1: Dynamic Pivot-based indexing**Input :** dataset DS ;maximal intersection α ;**Output:** set P of pivots with their assigned elements

```

1 if  $DS$  is the first source then
2    $I_c = 0$ ; // intersections records pivots
3    $N = 0$ ; // number of read records
4    $P = S$ ; // small set  $S$  of initial pivots with  $rad(p_i) = 0$ , for  $p_i \in P$ 
5 foreach  $x \in DS$  do
6    $N = N + 1$ ;
7    $minDist = \infty$ ;
8    $bestPivot = null$ ;
9   foreach  $p_i \in P$  do
10    if  $dist(x, p_i) \leq rad(p_i)$  then
11       $I_c = I_c + 1$ ;
12    if  $dist(x, p_i) < minDist$  then
13       $minDist = dist(x, p_i)$ ;
14       $bestPivot = p_i$ ;
15    $elem(bestPivot) = elem(bestPivot) \cup \{x\}$ ;
16   if  $rad(bestPivot) < minDist$  then
17      $rad(bestPivot) = minDist$ ;
18    $overlap(P) = \frac{I_c}{N \times |P|}$ ;
19   if  $overlap(p) > \alpha$  then
20     run Algorithm 2 // Generate new pivot

```

Let I_c be the sum of the number of pivot intersections for any record x , N the number of indexed objects, and $|P|$ the number of pivots. We define the *overlap factor* of the pivots as:

$$overlap(P) = \frac{I_c}{N \times |P|} \quad (2)$$

The overlap factor thus determines the average number of pivot intersections per record normalized by the total number of pivots. A low overlap value means that the data objects are largely partitioned according to their pivots, e.g., when the radii are relatively narrow. Additional data objects are assigned to the closest pivots so that the radii and thus the overlap between pivots tends to increase. This will also reduce the filter effects and thus increase the number of necessary match comparisons.

We use an adaptation parameter α to control the maximal overlap ratio between pivots, i.e., we increase the number of pivots as soon as this value is exceeded in order to increase the number of pivots for a growing number of data objects. Algorithm 1 specifies the dynamic indexing using parameter α . The algorithm starts with a small set of pivots P (line 4), that

Algorithm 2: Generate new pivot**Input** : set of pivots P with their elements;

```

1 choose  $p \in P$ ; // pivot with max cardinality
2 choose  $p_{new} = x \in elem(p)$ : furthest element from  $p$ ;
3 foreach  $p_i \in P$  do
4   foreach  $e \in elem(p_i)$  do
5     if  $dist(e, p_{new}) \leq dist(e, p_i)$  then
6        $elem(p_{new}) = elem(p_{new}) + \{e\}$ ;
7       store  $dist(e, p_{new})$ ;
8       update  $rad(p_{new})$ ; // if necessary
9        $elem(p_i) = elem(p_i) - \{e\}$ ;
10      update  $rad(p_i)$ ; // if necessary
11  $P = P + \{p_{new}\}$ ;

```

can be chosen for example randomly. In the next step, new records of the dataset are read sequentially and compared with the existing pivots. I_c is incremented each time the distance between x and any pivot p_i is smaller than the radius $rad(p_i)$ (line 10 and 11). Finally, x is added to the elements of the closest pivot and the radius of this pivot is possibly updated (lines 15-17). Furthermore, the overlap factor is calculated and compared with α (lines 18-20). If the overlap exceeds the value of α a new pivot p_{new} is generated (Algorithm 2) and all the objects already indexed are re-partitioned over the pivots.

Note that there are different strategies to select a new pivot (line 1 and 2 of Algorithm 2); e.g., choose the new pivot as the furthest record from the elements of the pivot with the highest cardinality (approach *MaxCard*) or from the pivot having the largest radius (*MaxRadius*). Another approach is to select the next pivot as the most furthest object to the already chosen pivots (*FurthestNode*). Preparatory experiments showed the highest effectiveness for *MaxCard* so that we will use this approach in our evaluation.

4.2 MP-PPRL using Dynamic Pivots

Running MP-PPRL using dynamic pivot based metric space is now straightforward and its steps are shown in Algorithm 3. For a set of data sources DS_i we start by indexing the first source DS_1 using algorithms 1 and 2. For each additional dataset DS_j $j \geq 2$ we run two methods successively: *link*(DS_j) which finds matches M_j between records of DS_j and records already assigned to $p_i \in P$ from former sources. The second method *index*(DS_j) partitions records of DS_j on the existing pivots $p_i \in P$ following the algorithms 1 and 2 which might lead to the generation of additional pivots.

This algorithm outputs pairs of similar records that form a similarity graph. By computing the connected components from this graph we generate initial clusters of matches. These

Algorithm 3: MP-PPRL in dynamic metric space

Input : $D = \{DS_j, j \in 1..i\}$ datasets;
 P set of pivots;
1 Output : C set of clusters;

2 $P = \emptyset$;
 3 $index(DS_1)$ // using algorithms 1 and 2
 4 **for** $j \geq 2$ **do**
 5 $link(DS_j)$;
 6 $index(DS_j)$;

clusters may still have several records per source and we will apply a post-processing steps to solve this problem.

In what follows we use the term *linkage-iteration* to denote the execution of $index(DS_i)$ on data source DS_i and $link(DS_{i+1})$ on the forthcoming data source DS_{i+1} .

5 Clustering

The goal of clustering is to group all matching records (Bloom filters) based on the previously determined pairs of matching records. Due to the assumption of duplicate-free sources, we have to ensure that every cluster should be *clean*, i.e., include at most one record per source. In this section we outline several approaches for early and late clustering that generate clean clusters during and after the linkage process, respectively.

5.1 Early Clustering

Early clustering algorithms build clean clusters progressively after each linkage-iteration by considering the linkage output of each iteration as a weighted bipartite graph. Based on some predefined criteria, edges are deleted from the graph so that each resulting cluster contains at most one record from any involved source. We describe two representative algorithms, Hungarian and Max-Both, of such a strategy.

Hungarian Algorithm: The Hungarian algorithm [Ku55] is a combinatorial optimization method to solve the assignment problem in polynomial time. The input of the algorithm is a weighted bipartite graph $G(S, T, E_w)$ with S and T consisting of two disjoint sets of vertices representing records from two different sources DS_i and DS_j and a set of weighted edges $E_w = \{(s, t) : s \in S \wedge t \in T\}$ where the weights represent the similarity values for pairs of elements of S and T . The goal of the Hungarian algorithm is to find an assignment with the highest global similarity between the elements of S and T so that each element from each set is linked with at most one element from the other set (1:1 mapping).

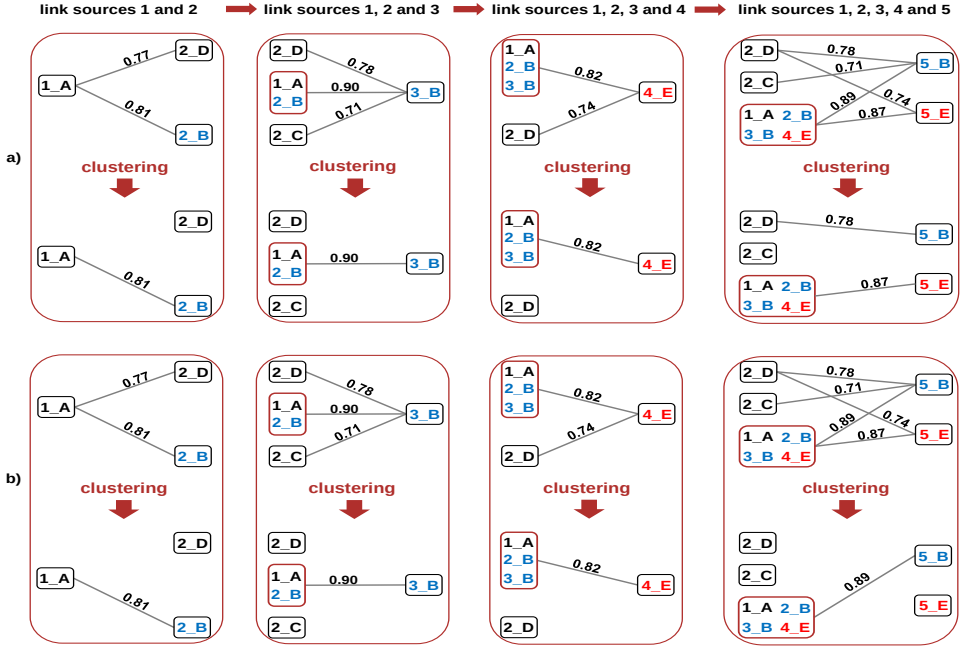


Fig. 3: Clustering the linkage result of 5 sources using the a) Hungarian Algorithm and b) Max-Both

For MP-PPRL the Hungarian algorithm is run after each linkage-iteration of a new dataset. Hence, after the linkage of the first two sources, DS_1 and DS_2 the algorithm is run in a straightforward manner on graph $G(DS_1, DS_2, E_e)$ and a set of clusters $c_i \in C$ with $1 \leq |c_i| \leq 2$ is generated. For each further source DS_i , $i \geq 3$, the linkage step computes the similarities between the elements $t \in DS_i$ and the clusters $c_i \in C$ and generates a new weighted bipartite graph $G(C, DS_i, E_w)$, that serves as an input for the Hungarian algorithm. The top of Fig. 3 shows the linkage of 5 sources and the application of the Hungarian algorithm to cluster the results. A perfect clustering would output the following clusters: $c_1 = \{2_B, 3_B, 5_B\}$, $c_2 = \{4_E, 5_E\}$, and the singletons $c_3 = \{1_A\}$, $c_4 = \{2_C\}$, $c_5 = \{2_D\}$ (colored with blue, red and black respectively).

Max-Both: Max-Both was introduced in [MGR02, DR02] to solve the multimapping problem of two sources. Max-Both works in a similar manner as the Hungarian algorithm; in each linkage-iteration it finds the 'best' assignment in a weighted bipartite graph. The only difference is that Max-Both keeps a link (s, t) only if this link is the best for both elements s and t . Therefore, for a bipartite graph $G(S, T, E_e)$ Max-Both starts by selecting for each element $s \in S$ the element $bestMath(s) = t \in T$ so that $sim(s, t) > sim(s, t') : \forall t' \in T$ and $t' \neq t$. On the other side, for each $t \in T$ the element $bestMath(t) = s \in S$ is selected so that $sim(s, t) > sim(s', t) : \forall s' \in S$ and $s' \neq s$. Then a mapping (s, t) is returned only if $bestMatch(s) = t \wedge bestMatch(t) = s$.

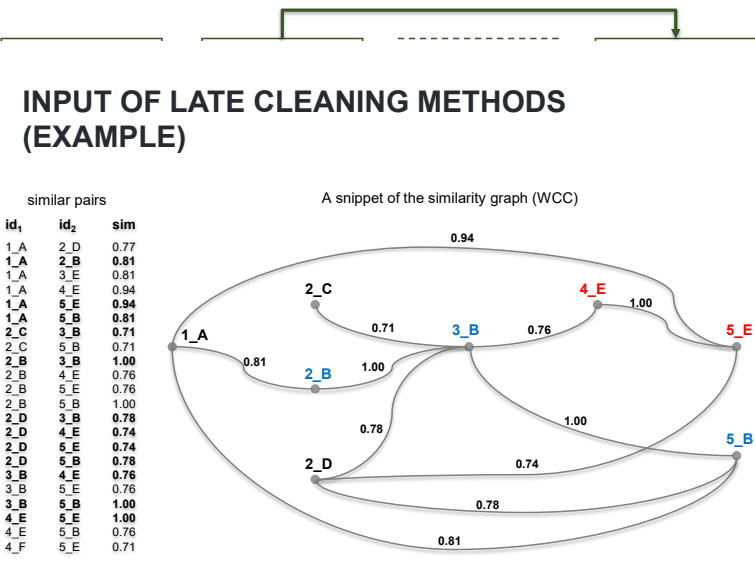


Fig. 5: Transformation of similar pairs to a similarity graph (weighted connected component). For clarity only the edges for the bold entries on the left side are shown.

The Bottom of Fig. 3 displays the output of Max-Both. Due to the simplicity of the input data Max-Both and the Hungarian algorithm return the same results in the first three iterations. In the last iteration, however, Max-Both keeps the best edge (higher similarity) between the large cluster and the singleton 5_B and prune all other edges. This leads to a more homogeneous cluster compared to the Hungarian algorithm, which try to maximize the weights by preserving a larger number of edges.

5.2 Late Clustering

In this approach the linkage process is run between records of all sources to generate a similarity graph. Clustering is performed late and uses a weighted connected component (WCC) as input instead of a weighted bipartite graph. The rationale behind this approach is to preserve a global view of the linkage result and avoid pruning edges in early iterations that might be good in later ones. Fig. 4 shows the steps of late clustering. First, the pairs of similar records from all sources are used to build WCCs, an example of such transformation is shown in Fig. 5. Then an optional step to split WCCs is run to reduce the size of large CCs, and thus reduce the complexity for clustering. To split connected components we can use either the method introduced in [Sa18] by deleting so-called weak links (the records connected with such links have at least one higher-similarity link to another record of the respective other source) inside each CC, or use the k-medoid algorithm to group a set of

a) sorting according to the avg. similarity			b) sorting according to the number of edges		
order	cluster	avg. sim	order	cluster	# edges
1	[2_B, 3_B, 5_B]	1.0	1	[1_A, 2_B, 3_B, 4_E, 5_B]	10
2	[2_B, 5_B]	1.0	2	[1_A, 2_B, 3_B, 4_E, 5_E]	10
3	[2_B, 3_B]	1.0	3	[1_A, 2_D, 3_B, 4_E, 5_E]	10
4	[3_B, 5_B]	1.0	4	[1_A, 2_D, 3_B, 4_E, 5_B]	10
5	[4_E, 5_E]	1.0	5	[1_A, 2_B, 3_B, 5_B]	6
6	[1_A, 4_E, 5_E]	0.96	6	[2_B, 3_B, 4_E, 5_B]	6
⋮			⋮		
67	[2_C, 3_B, 4_E]	0.74	67	[4_E, 5_B]	1
68	[2_D, 5_E]	0.74	68	[2_D, 5_E]	1
69	[2_D, 4_E]	0.74	69	[2_D, 4_E]	1
70	[2_C, 5_B]	0.71	70	[2_C, 5_B]	1
71	[2_C, 3_B]	0.71	71	[2_C, 3_B]	1
72	[4_F, 5_E]	0.71	72	[4_F, 5_E]	1

Fig. 6: The outputs of the algorithm Sort and Keep Best on the similarity graph of Fig. 5 using (a) avg. similarity and (b) number of edges. Returned clusters for each sorting criteria are green framed.

UNIVERSITÄT
LEIPZIG Database Group

records into clusters of similar objects. In the following, we present two late clustering methods, graph multicut and SKB (sort and keep best).

Graph Multicut: Multicut is a graph partitioning problem that takes as input a connected graph $G(V, E)$, a weight function $w : E \rightarrow R$ that assigns weights to the edges E , and a set of k pairs $(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)$. The algorithm tries to find the set of edges $F \subset E$ whose removal disconnect each pair (s_i, t_i) for $i = 1 \dots k$. A minimum graph multicut returns the set F with the lowest cost. This problem can be defined as a linear program:

$$\begin{aligned}
 & \text{minimize} && \sum_{e \in E} w_e \times d_e \\
 & \text{subject to} && \sum_{e \in p} d_e \geq 1 \quad \forall p \in P_{s_i, t_i}, i = 1 \dots k \\
 & && d_e \geq 0 \quad \forall e \in E
 \end{aligned} \tag{3}$$

This linear program assigns the smallest positive value d_e to each edge $e \in E$ which is on any path p_i that joins the pairs s_i, t_i (pairs to be disconnected) so that the sum of the assigned values d_i for each path is greater than 1. The edges to be removed (cut edges) are contained in the set $F = \{e \in E : d_e \geq 0.5\}$ [GVY93].

This algorithm can be easily adapted to cluster the result of MP-PPRL. Each WCC constitutes a graph $G_i(V_i, E_i)$ of records V_i and weighted edges E_i (similarity values). We consider each pair of records $x, y \in V_i$ from the same source as terminals to be disconnected, and generate clusters by solving linear program 3. For the example graph in Fig. 5 the pair $(5_E, 5_B)$ from the fifth source constitutes two terminals to be disconnected. Multicut algorithm assigns to each edge of the paths connecting 5_E and 5_B the smallest possible value d_i then prunes all edges having a value ≥ 0.5 . Applying Multicut on the graph of Fig. 5 returns the following clusters: $\{1_A, 2_B, 3_B, 4_E, 5_E\}$, $\{2_D, 4_F, 5_B\}$ and

Algorithm 4: Sort and Keep Best (SKB)**Input** : WCC weighted connected component; K sorting criteria;

```

1 Output :  $C$  set of clean clusters;;
2  $C = \emptyset$ ;
3  $tmp = \emptyset$ ; // to store records of clean clusters
4 generate all possible clean clusters  $c_i$ ;
5 sort  $c_i$  according to  $K$ ;
6 foreach  $clr \in c_i$  do
7   if  $clr \cap tmp = \emptyset$  then
8      $C = C + \{clr\}$ ;
9      $tmp = tmp \cup clr$ ; // add records of  $clr$  to  $tmp$ 
10  else
11     $\perp$  prune  $clr$ ;

```

the singleton $\{2_C\}$. As we can see, Multicut is a kind of generalisation of the Hungarian algorithm for multiple sources which tries to keep as many edges as possible.

Sort and Keep Best (SKB): Algorithm 4 implements this method. It takes as input a WCC and starts to generate all possible clean clusters contained in it. Based on some criteria, the algorithm sorts the generated clusters. Finally, the clusters are read sequentially and it is checked whether the actual cluster includes records that are contained in better (already processed) clusters or not (line 7). If this is not the case the cluster is added to the set of final clusters and its elements are stored to check forthcoming clusters (line 8-9).

We evaluated two criteria to sort the clusters: 1) Average similarity inside the cluster $Avg_sim = \sum_{e \in E_c} w_e / |E_c|$, if Avg_sim of two clusters are equal then sort according to $|E_c|$. 2) The number of edges $|E_c|$ between the elements of a cluster, if $|E_c|$ of two clusters are equal then sort according to Avg_sim . Fig. 6 illustrates the output of both sorting criteria on the similarity graph of Fig. 5. Sorting the clusters by Avg_sim leads generally to small but more homogeneous clusters than sorting them by edges. The latter method tends to create large clusters that might include miss-matches.

6 Evaluation

After the description of the experimental setup, we evaluate the performance of the proposed dynamic selection of pivots in metric space for MP-PPRL. In particular, we analyse the influence of the overlap parameter α and present a comparison with Sparse Spatial Selection (SSS), an alternate dynamic pivoting scheme. Furthermore, we conduct a comparative analysis of the match quality and runtimes of the presented early and late clustering strategies.

# sources	dataset	1	2	3	4	5	6	7	8	9	10
3	S-NCVR	58%	17%	25%							
	R-NCVR	20%	50%	30%							
5	S-NCVR	60%	3%	5%	7%	25%					
	R-NCVR	19%	24%	24%	19%	14%					
7	S-NCVR	60%	1%	2%	3%	4%	5%	25%			
	R-NCVR	16%	21%	21%	16%	11%	8%	7%			
10	S-NCVR	61%	1%	1%	1%	1%	2%	2%	3%	3%	25%
	R-NCVR	15%	20%	20%	15%	10%	6%	5%	4%	3%	2%

Tab. 1: Distributions of duplicate records over the sources to be matched for both dataset collections S-NCVR and R-NCVR.

6.1 Experimental Setup

For our experiments we use two collections of datasets with different sizes and number of sources (parties) to simulate a MP-PPRL process. The first collection, S-NCVR, was generated from the publicly available North Carolina Voter Register (NCVR) dataset. Using a snapshot of about 7 million persons several sources have been generated. Duplicate records over the different sources have been created by introducing some modifications (typos) to the original attribute values. The second collection R-NCVR, also obtained from NCVR, represents real data without any modification. Duplicate records over sources arise from real changes or modification in the attributes values of some persons. Table 1 shows the number of sources and the distribution of duplicate records over the sources for both sizes 100,000 and 500,000. The two collections have different duplicates distributions, e.g., to link three parties each containing 100,000 records, the three sources from S-NCVR contains about 60% of singletons, 17% of records are duplicate in two sources and 25% of records are present in all three sources. The distribution of sources from R-NCVR decreases the number of duplicates over sources when the number of sources increase. Note that the duplicates are a mixture of similar and equal records distributed over the sources.

Records of both collections of data have been encoded to Bloom filters of length 1,000 using between 10 and 30 hash functions. All experiments are conducted single-threaded on a machine with a 4-core 4.00GHz CPU and 32 GB of main memory.

6.2 Evaluation of Dynamic Pivoting

6.2.1 Influence of max overlap value α

We use the first collection of datasets R-NCVR to determine the influence of parameter α and to determine effective settings for it. We run MP-PPRL without any clustering method to

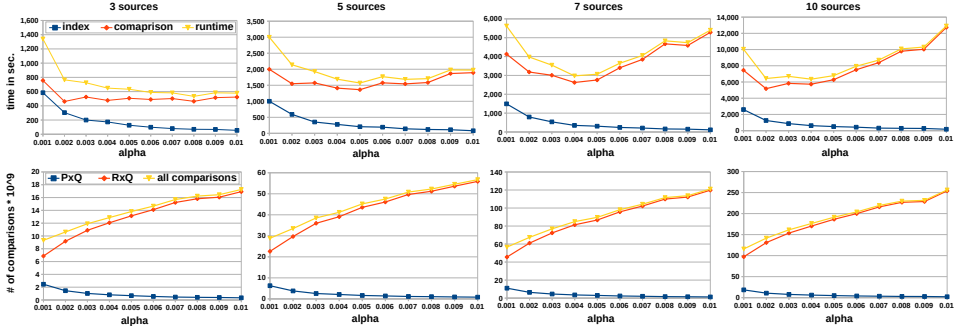


Fig. 7: Runtimes (index and comparison) and number of comparisons w.r.t. the value of α to link 3,5,7 and 10 sources each containing 100,000 records.

link sets of sources with different configuration w.r.t the number of sources and the size of each source. We varied the value of α between 0.001 and 0.01 and set the similarity threshold to 0.75. For each configuration we evaluate the time needed to index records, i.e., to assign them to their pivots, and to find the matches. Furthermore, we analyze the number of distance computations.

The top of Fig. 7 shows the index, comparison and total runtime to link sets of sources (3, 5, 7 and 10 sources) each containing 100,000 records. We observe that small α values < 0.002 lead to both high index and comparison times. For $\alpha = 0.001$ the index time, i.e., the time to dynamically generate new pivots and to distribute records on these pivots, represents about 43% of the total runtime when MP-PPRL is run for three sources. The reason of this high index time is that low α values are frequently exceeded leading to the determination of additional pivots and a corresponding reassignment of record to these pivots. Furthermore, the generated pivots need to be compared with all the query records from the second source to check the triangle inequality, which increases comparison time. For $\alpha = 0.001$ the number of dynamically generated pivots from the first source is about 10,000 pivots that must be compared with the 100,000 records from the second source before the real linkage begins.

On the other side, higher α values decrease indexing time because fewer pivots are generated. However, the comparison time increases dramatically when $\alpha > 0.005$ and the number of sources to link is greater than five. For such α values a large number of records are distributed on a small number of pivots in the index phase which causes an enlargement of the pivots radii and therefore their ability to exclude pairs from farther comparison using the triangle inequality. This can be shown in the bottom of Fig. 7, where the number of distance computation to link ten sources grows from 97×10^9 to 254×10^9 when α is changed from 0.001 to 0.01.

Another behaviour we can observe is that the comparison time is not always related to the number of comparisons computed. By assigning α small values we reduce in fact the

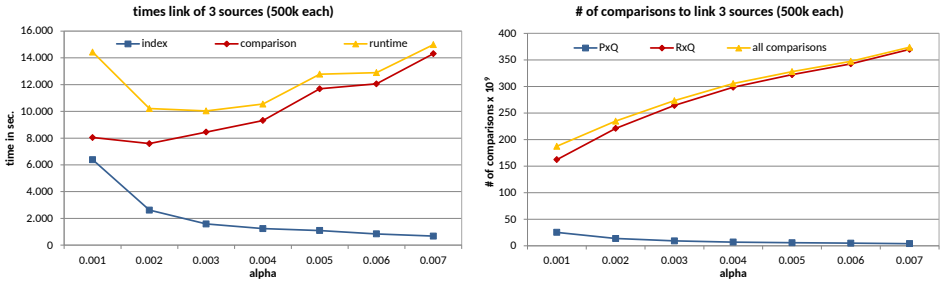


Fig. 8: Runtimes (index and comparison) and number of comparisons w.r.t. the value of α to link 3 sources each containing 500,000 records.

number of comparisons (queries with pivots + queries records), however, the computation time does not follow this trend. This is due to the overhead of parsing all queries for each pivot and the distance function (XOR) that is known to be very cheap to execute.

To investigate the best α value for large dataset we run MP-PPRL to link a set of three sources each containing 500,000 records and varying α between 0.001 and 0.007. The right part of Fig. 8 shows the index, comparison and total runtime. For this experiment we observe the same behaviour as for smaller datasets, i.e., very small α values ≤ 0.002 increase both the index and comparison time, and large α values ≥ 0.003 decrease the index time but increase the comparison time for the same reason as mentioned above. As we can see the best runtime (index + comparison) is obtained for α values between 0.002 and 0.004. We use $\alpha = 0.003$ in the following experiments.

A very high value of parameter α , e.g., 1, means that the generation of new pivots will never be triggered which corresponds to a static pivoting approach where the initially selected pivots are not changed any more. The shown curves in Fig. 7 and 8 show that the runtimes and number of comparisons for the highest α values are much worse than for the best settings of α which underlines the high value of the proposed dynamic pivoting approach.

6.2.2 Comparison with Sparse Spatial Selection (SSS) Method

We now compare the performance of our method of dynamically selecting pivots with the the Sparse Spatial Selection method (SSS) that was proposed outside the context of PPRL. SSS starts with a random record x as pivot. The next pivots are records having a distance $\geq M \times \beta$ to any already selected pivot, being M the maximal distance between any two records in the dataset and β a constant parameter taking values in $[0.35, 0.40]$. We run some initial experiments and found $\beta = 0.54$ the optimal value for our datasets. To compare our method with SSS, we run MP-PPRL to link datasets of 3 and 7 sources each containing 500,000 records from the second collection, S-NCVR. Table 2 shows the achieved results for our method (named *overlap*) and SSS for different similarity threshold (0.75 – 0.95).

		0.75		0.8		0.85		0.9		0.95	
		SSS	overlap	SSS	overlap	SSS	overlap	SSS	overlap	SSS	overlap
3 src.	# Pivot_1	17,064	6,247	17,064	6,247	17,064	6,247	17,064	6,247	17,064	6,247
	# Pivot_2	21,436	7,276	21,436	7,276	21,436	7,276	21,436	7,276	21,436	7,298
	P x Q ($\times 10^9$)	19.25	6.76	19.25	6.76	19.25	6.76	19.25	6.76	19.25	6.77
	R x Q ($\times 10^9$)	175.90	228.83	37.04	62.71	7.69	14.82	1.91	3.91	0.48	1.06
	index time	35	16	32	16	26	15	26	15	29	15
	runtime	179	153	102	72	85	57	57	46	51	54
7 src.	# Pivot_1	16,825	6,279	16,825	6,279	16,825	6,279	16,825	6,279	16,825	6,279
	# Pivot_2	30,742	9,724	30,742	9,725	30,742	9,726	30,742	9,741	30,742	9,835
	P x Q ($\times 10^9$)	74.33	24.64	74.33	24.64	74.33	24.64	74.33	24.65	74.33	24.82
	R x Q ($\times 10^9$)	1,235.27	1,412.03	265.37	366.05	55.57	85.53	13.92	22.77	3.50	6.13
	index time	48	37	47	39	33	31	33	47	34	38
	runtime	1,237	1,185	545	412	309	176	259	162	292	152

Tab. 2: Comparison of our method (overlap) with SSS to link 3 and 7 sources from the S-NCVR each containing 500,000 records. We investigate the number of pivots generated in the first (# pivot_1) and last (# pivot_2) linkage iteration, the number of comparisons between pivots and queries (P x Q) and between indexed records and queries (R x Q), the index time and the complete runtime in minutes. (best values in bold)

We report the number of generated pivots, the number of comparisons between pivots and queries (P x Q) and between indexed records and queries (R x Q). Furthermore, we consider the index time needed to find adequate pivots and partition records over them and the complete runtime.

We observe that SSS generates many more (three times more) pivots as our overlap-based method. To index the first source (for both 3 and 7 sources) SSS needs about 17,000 pivots while our method generates only about 6,000 pivots for the same source. During the MP-PPRL process SSS continues to generate numerous pivots whose number reach 30,000 to compare 7 sources. While the high number of pivots generally can reduce the number of comparisons between indexed records and queries (as explained before) it leads, however, to a large index time and runtime, and a high number of comparisons between queries and pivots. For both number of sources (3 and 7) SSS requires three times more comparisons between pivots and queries as our method (6.76×10^9 vs. 19.25×10^9 for 3 sources). Hence, our methods outperforms SSS w.r.t. the quality and number of pivots generated and the runtime of MP-PPRL.

6.3 Comparison of Clustering Methods

6.3.1 Number of sources

We first evaluate the quality of the proposed early and late clustering approaches, Hungarian algorithm (HUNG), Max-Both (MAX-B), graph multicut (M-CUT) and sort and keep best using average similarity (SKB-S) and number of edges (SKB-E) as sorting criterion, as well as of the baseline approach connected components (CC) in terms of precision, recall and f-measure for different number of sources. In the first experiment, we use sources of

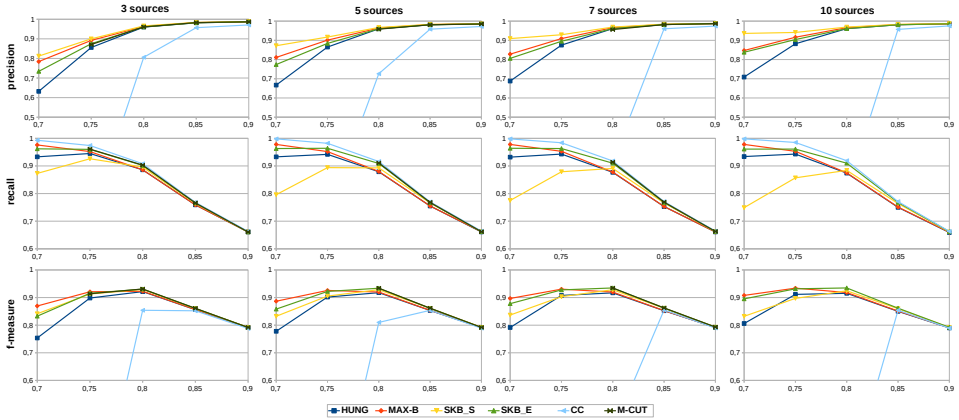


Fig. 9: Quality of the different clustering methods run on 3,5,7 and 10 sources of size 100,000 from the S-NCVR collection and varying the similarity threshold between 0.7 and 0.9

size 100,000 and vary the numbers of sources to be linked between 3 and 10. For each clustering algorithm the similarity threshold is varied between 0.7 and 0.9. Fig. 9 shows the results of the experiment using datasets from the S-NCVR collection. As we can see, clusters generated by building connected components from the similarity pairs are generally not usable, especially for low threshold. The precision of such clusters by threshold ≤ 0.75 is about 0, while the recall is not considerably higher than the other clustering methods.

CC is clearly outperformed by the proposed early and late clustering schemes. The best f-measure values are generally achieved for similarity values between 0.75 and 0.8. The Hungarian algorithm generally achieves the lowest f-measure due to its low precision compared to the other approaches. Max-Both by contrast achieves a much better precision and is among the best performing approaches, especially for lower similarity thresholds. The f-measure results for the late clustering approaches are relatively close together. However, graph multicut cannot be applied for low thresholds (≤ 0.7) or high number sources (10 sources) due to its high runtime and memory consumption. The late clustering approach SKB-S, that elects clusters with the highest similarity, achieves the best precision especially for lower similarity. Interestingly, the f-measure does not decrease when the number of sources is increased. This is surprising since it is generally more difficult to correctly find bigger clusters than smaller clusters. This has been possible despite the existence of many large clusters for this synthetic dataset where 25% of all duplicate records belong to the largest clusters with records from all sources (Table 1).

To study the impact of a different duplicate distributions with only few large clusters we run a similar experiment on the second collection of datasets, R-NCVR, for 3 and 10 sources of size 100,000. As we can see in Fig 10 the CC method has again the poorest cluster quality due to a very low precision for lower similarity thresholds ≤ 0.75 . Now the late clustering

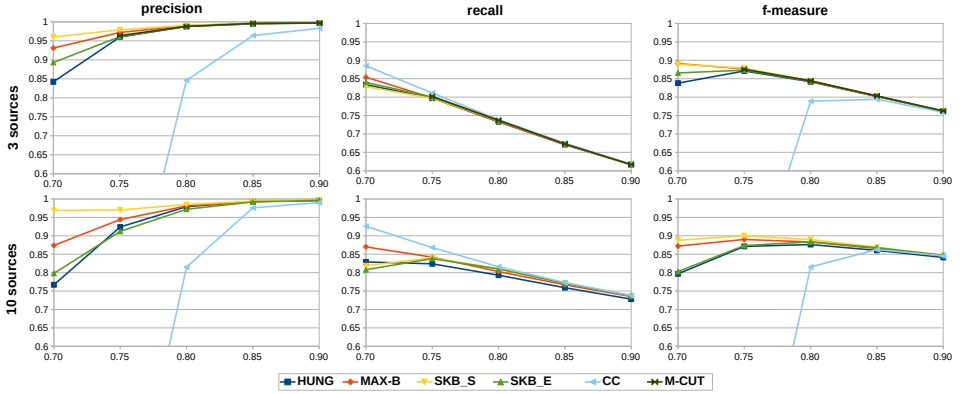


Fig. 10: Quality of the different clustering methods run on 3 sources and 10 sources of size 100,000 from R-NCVR

method SKB-S performs best since it achieves not only the best precision but can achieve a similarly good recall than the other late clustering methods. Max-Both finds a good balance between precision and recall and performs almost as good as SKB-S. Again, the f-measure for the larger number of sources (10 sources) is similar to the one for only 3 sources.

6.3.2 Size of sources

Figure 11 shows the precision, recall and f-measure results and the total runtime (linkage and clustering) for 5 sources and 7 sources applying a similarity threshold of 0.75. The size of each source is set to 100,000 and 500,000 for both number of sources. As we can see, scaling the size of the sources leads to a drop of the quality for all clustering methods. Precision of the Hungarian algorithm drops the most by about 10% from 0.95 to 0.85 followed by Max-Both by about 5%. Only SKB-S shows a small decrease by 1% in precision when scaling up the size of sources from 100,000 to 500,000 records. Furthermore, SKB-S is the only method that still manifests a precision above 0.9 for the larger datasets. This is due to its selecting clusters with the highest intra-cluster similarity as clean clusters. However, this also reduces recall since such a high similarity is typically reached by smaller clusters that are thus preferred over larger clusters with lower internal similarity. Hence, SKB-S achieves the lowest recall (0.79) for 5 sources of size 500,000 records. Note that SKB-E, which promotes large clusters over smaller ones, achieves a similar recall as SKB-S. This is because R-NCVR contains more small clusters of size 2 and 3 than larger ones. Among all algorithms Max-Both returns the best recall for all sizes and number of source. The f-measure of SKB-S and Max-Both are similar with a light advantage for SKB-S.

The right of Fig. 11 shows the runtime to run both linkage and post-processing steps. All four methods achieve similar runtime of about 25 and 50 minutes to process 5 sources and

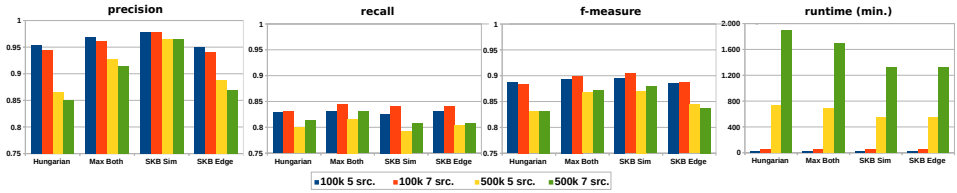


Fig. 11: Quality of the clustering methods and total runtime of linkage process for similarity threshold 0.75 compared with relation to the size and number of sources from R-NCVR

7 sources with 100,000 records each. However, scaling the size of the sources increases the total runtime by a factor of 10 for all methods. This is due the use of metric space for the linkage. As explained in section 3.3 metric space finds all pairs of records that have a similarity above a predefined threshold. We observe, however, that both SKB methods achieve generally the lowest runtime.

7 Conclusions

We studied the use of metric space for multi-party privacy preserving records linkage (MP-PPRL) to efficiently link and cluster records encoded as Bloom filters. We proposed a dynamic pivot-based metric space approach to reduce the number of comparisons that can adapt the number and choice of pivots for a growing number of data sources and thus increasing data volume. The approach is driven by a parameter to control and limit the overlap between the pivots in the metric space. The evaluation showed that this method is very efficient to link multiple sources. Furthermore we presented five early and late clustering methods that create clusters containing at most one element from each source. Early clustering approaches build clusters during the linkage process and late clustering postpone the determination of clusters after all sources have been linked. Our evaluation shows the high scalability and good quality of Max-Both as an early clustering method and SKB-S as a late clustering method.

Despite the effectiveness of the dynamic pivot-based metric space the runtime of the new approaches still increase more than linear with data size. We will thus analyze further runtime improvements such as the adoption of parallel processing on frameworks such as Apache Spark and the combined use of metric space and blocking.

Bibliography

- [B170] Bloom, Burton H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Commun. ACM, 13(7):422–426, July 1970.
- [BRF15] Boyd, James H.; Randall, Sean M.; Ferrante, Anna M.: Application of Privacy-Preserving Techniques in Operational Record Linkage Centres. In: Medical Data Privacy Handbook. Springer, pp. 267–287, 2015.

- [Ch12] Christen, P.: *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, 2012.
- [DR02] Do, H.; Rahm, E.: COMA: A System for Flexible Combination of Schema Matching Approaches. In: *Proc. VLDB conf.* pp. 610–621, 2002.
- [Du12] Durham, E.A.: *A framework for accurate, efficient private record linkage*. PhD thesis, Faculty of the Graduate School of Vanderbilt University, Nashville, TN, 2012.
- [Fr18] Franke, M.; Sehili, Z.; Gladbach, M.; Rahm, E.: Post-processing Methods for High Quality Privacy-Preserving Record Linkage. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. 2018.
- [FSR18] Franke., M.; Sehili., Z.; Rahm., E.: Parallel Privacy-preserving Record Linkage using LSH-based Blocking. In: *Proc. 3rd Int. Conf. on Internet of Things, Big Data and Security (IoTBDs)*. INSTICC, SciTePress, pp. 195–203, 2018.
- [Gi16] Gibberd, A.; Supramaniam, R.; Dillon, A.; Armstrong, B.; O’Connell, D.: Lung cancer treatment and mortality for Aboriginal people in New South Wales, Australia: Results from a population-based record linkage study and medical record audit. *BMC Cancer*, 16, 12 2016.
- [Gl18] Gladbach, M.; Sehili, Z.; Kudrass, T.; Christen, P.; Rahm, E.: Distributed Privacy-Preserving Record Linkage Using Pivot-Based Filter Techniques. In: *Proc. ICDE workshops (ICDEW)*. pp. 33–38, April 2018.
- [GVY93] Garg, N.; Vazirani, V. V.; Yannakakis, M.: Approximate Max-Flow Min-(multi)cut Theorems and Their Applications. *SIAM Journal on Computing*, 25:698–707, 1993.
- [Ku55] Kuhn, H.W.: The Hungarian Method for the Assignment Problem. *Naval Res. Logist. Quart.*, 2:83–98, 01 1955.
- [Ku11] Kuehni, Claudia E; Rueegg, Corina S; Michel, Gisela; Rebholz, Cornelia E; Strippoli, Marie-Pierre F; Niggli, Felix K; Egger, Matthias; von der Weid, Nicolas X; for the Swiss Paediatric Oncology Group (SPOG): Cohort Profile: The Swiss Childhood Cancer Survivor Study. *Int. Journal of Epidemiology*, 41(6):1553–1564, 10 2011.
- [MGR02] Melnik, S.; Garcia-Molina, H.; Rahm, E.: Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: *Proc.18th Int. Conf. on Data Engineering*. pp. 117–128, Feb 2002.
- [PB07] Pedreira, Oscar; Brisaboa, Nieves R.: Spatial Selection of Sparse Pivots for Similarity Search in Metric Spaces. In: *SOFSEM 2007: Theory and Practice of Computer Science*. Springer Berlin Heidelberg, pp. 434–445, 2007.
- [Sa18] Saeedi, A.; Nentwig, M.; Peukert, E.; Rahm, E.: Scalable Matching and Clustering of Entities with FAMER. *CSIM Quarterly*, 16:61–83, 2018.
- [SBR09] Schnell, R.; Bachteler, T.; Reiher, J.: Privacy-preserving record linkage using Bloom filters. *BMC Medical Informatics and Decision Making*, 9(1):41, Aug 2009.
- [SBR11] Schnell, R.; Bachteler, T.; Reiher, J.: A Novel Error-Tolerant Anonymous Linking Code. Technical Report WP-GRLC-2011-02, German Record Linkage Center, Duisburg, 2011.

- [Se15] Sehili, Z.; Kolb, L.; Borgs, C.; Schnell, R.; Rahm, E.: Privacy Preserving Record Linkage with PPJoin. In: Proc. BTW. pp. 85–104, 2015.
- [SR16] Sehili, Z.; Rahm, E.: Speeding up Privacy Preserving Record Linkage for Metric Space Similarity Measures. *Datenbank-Spektrum*, 16(3):227–236, Nov 2016.
- [Tr00] Traina, C.; Traina, A.; Seeger, B.; Faloutsos, C.: Slim-Trees: High Performance Metric Trees Minimizing Overlap between Nodes. In: *Advances in Database Technology — EDBT 2000*. Springer, pp. 51–65, 2000.
- [Va17] Vatsalan, D.; Sehili, Z.; Christen, P.; Rahm, E.: Privacy-preserving record linkage for big data: Current approaches and research challenges. In: *Handbook of Big Data Technologies*, pp. 851–895. Springer, 2017.
- [VCR20] Vatsalan, D.; Christen, P.; Rahm, E.: Incremental clustering techniques for multi-party Privacy-Preserving Record Linkage. *Data & Knowledge Engineering*, 2020.
- [VCV13] Vatsalan, D.; Christen, P.; Verykios, V. S.: A taxonomy of privacy-preserving record linkage techniques. *Information Systems*, 38(6):946–969, 2013.
- [Xi08] Xiao, C.; Wang, W.; Lin, X.; Yu, J. X.: Efficient Similarity Joins for Near Duplicate Detection. In: *Proc. 17th Int. Conf. on World Wide Web*. pp. 131–140, 2008.
- [Ze06] Zezula, P.; Amato, G.; Dohnal, V.; Batko, M.: *Similarity search: the metric space approach*. Springer, 2006.

Towards Resilient Data Management for the Internet of Moving Things

Elena Beatriz Ouro Paz,¹ Eleni Tzirita Zacharatou,² Volker Markl³

Abstract: Mobile devices have become ubiquitous; smartphones, tablets and wearables are essential commodities for many people. The ubiquity of mobile devices combined with their ever increasing capabilities, open new possibilities for Internet-of-Things (IoT) applications where mobile devices act as both data generators as well as processing nodes. However, deploying a stream processing system (SPS) over mobile devices is particularly challenging as mobile devices change their position within the network very frequently and are notoriously prone to transient disconnections. To deal with faults arising from disconnections and mobility, existing fault tolerance strategies in SPS are either checkpointing-based or replication-based. Checkpointing-based strategies are too heavyweight for mobile devices, as they save and broadcast state periodically, even when there are no failures. On the other hand, replication-based strategies cannot provide fault tolerance at the level of the data source, as the data source itself cannot be always replicated. Finally, existing systems exclude mobile devices from data processing upon a disconnection even when the duration of the disconnection is very short, thus failing to exploit the computing capabilities of the offline devices. This paper proposes a buffering-based reactive fault tolerance strategy to handle transient disconnections of mobile devices that both generate and process data, even in cases where the devices move through the network during the disconnection. The main components of our strategy are: (a) a circular buffer that stores the data which are generated and processed locally during a device disconnection, (b) a query-aware buffer replacement policy, and (c) a query restart process that ensures the correct forwarding of the buffered data upon re-connection, taking into account the new network topology. We integrate our fault tolerance strategy with NebulaStream, a novel stream processing system specifically designed for the IoT. We evaluate our strategy using a custom benchmark based on real data, exhibiting reduction in data loss and query runtime compared to the baseline NebulaStream.

Keywords: Mobile Stream Processing, Internet-of-Things, Fault Tolerance, Buffering

1 Introduction

Existing research and development for the Internet-of-Things (IoT) has primarily focused on stationary objects that are associated with a fixed location. However, some of the most important pieces in the IoT landscape are devices that can move (i.e. dynamically change their geo-spatial position), such as mobile phones and tablets. Mobile devices have become ubiquitous; smartphones, tablets and wearables have all become daily commodities for many people. A report published by GSMA in 2020 estimates that by 2025 there will be 5.8 billion

¹ Teradata (this work was done while the author was at TU Berlin, Germany), elenabeatriz.ouropaz@teradata.com

² TU Berlin, Germany, eleni.tziritazacharatou@tu-berlin.de

³ TU Berlin, DFKI GmbH, Germany, volker.markl@tu-berlin.de

mobile subscribers and that nearly 80% of connections will be made from smartphones [GS]. The ubiquity of mobile devices combined with their ever increasing computing capabilities give rise to new systems that leverage mobile devices in the processing of streaming data. Furthermore, they enable new IoT applications where mobile devices are used to both generate and process data. While mobile devices open numerous possibilities for the IoT, using them to process data is challenging. Mobile devices are notoriously prone to transient disconnections due to network instabilities while their near-constant mobility within the network renders much of the existing research in the field of stream processing inapplicable.

State-of-the-art stream processing systems that use mobile devices base their fault tolerance strategies on replication [OSP18, CS20] or checkpointing [WP14, MRH14]. Neither of these techniques is optimal for using mobile devices as data sources that also perform processing. Checkpointing-based techniques require the devices to store and broadcast snapshots of their state periodically. This is a pessimistic approach that introduces a considerable overhead irrespective of whether failures occur or not, and can drain the scarce resources of mobile devices. Replication-based techniques, on the other hand, replicate query operators across multiple nodes in the network. However, replicating the source can be impractical or costly, as it requires to have redundant sensors capturing the same input. When redundant sensors are not available, replication-based techniques cannot provide fault tolerance for data sources. Finally, existing approaches assume that disconnections are permanent and exclude the disconnected device from data processing. As a result, any data generated during the disconnection are lost. Furthermore, when the device re-connects, it is treated as a new node and all the queries that use it as a source have to be redeployed. This imposes an undue performance overhead when the disconnection is only transient.

In this paper, we propose a fault tolerance strategy for overcoming transient disconnections of mobile devices that both generate and process data in a streaming system. We particularly focus on transient disconnections caused by mobility, where the device might move through the network during the disconnection period, and then re-connect at a different point in the network. In addition, we assume that we do not have control over the data source, and therefore we cannot pause and restart the source or change the data generation rate. The goal of our proposed strategy is twofold: (a) avoid losing data during the disconnection, and (b) resume query execution quickly and consistently when the device re-connects. To achieve this goal, we employ the following mechanisms. First, we propose a data logging mechanism that enables mobile devices to keep processing the data they generate while they are disconnected by temporarily storing the processed data in a circular buffer. That way, we leverage the idle disconnection period for computation, and reduce, or completely avoid, data loss. Furthermore, we propose a query restart process that achieves efficient and consistent resumption of query execution, even when mobile devices move during the period of disconnection and reappear at a new point in the network. Our query restart process examines whether the device can still reach its previously assigned downstream node after the re-connection, and only redeploys queries when the downstream node cannot be reached. When a query is redeployed, the device may be assigned different tasks than

those it was originally performing. Our restart process handles this situation by generating new paths through which the buffered data can be forwarded directly to the pertaining node, when needed. This is achieved by taking advantage of the natural way in which stream processing systems model queries as directed acyclic graphs (DAGs). Our contributions can be summarized as follows:

- We mitigate the impact of transient disconnections of mobile devices with an end-to-end reactive fault tolerance strategy that leverages the offline processing capabilities of the devices. In our strategy, mobile devices continue processing the data they generate locally during a disconnection, and store the processed data in a circular buffer until connection is regained.
- We propose a query-aware replacement policy to make space in the buffer when needed.
- We develop a query restart process that addresses the mobility of devices during transient disconnections.
- We integrate our solution with the NebulaStream (NES) platform [Ze20a], a novel streaming system for the IoT. We call our solution NebulaStream-MSS, where *MSS* stands for *Mobile Source Support*.
- We evaluate our solution on real data collected from a sensor attached on a football player during a match. Our results show that buffering reduces data loss by 63% over a disconnection period of 30 seconds compared to the vanilla NES. Additionally, when several disconnections occur during query execution, our restart process reduces the query runtime by more than 2× by avoiding query redeployment whenever possible.

We envision our approach as a building block that can be combined with other fault tolerance strategies to handle other failure scenarios, such as devices that permanently exit the network, or failures occurring in nodes that are not data sources.

In the remainder of this paper, we first motivate our work in Section 1.1 and then present the background concepts that lay the foundation for NebulaStream-MSS in Section 2. Section 3 describes our approach, which we then experimentally evaluate in Section 4. Finally, we present an overview of related work in Section 5 before concluding and discussing future work in Section 6.

1.1 Motivational IoT Application Scenario

To illustrate the importance of a fault tolerance strategy tailored for mobile data sources, let us look at an IoT disaster management application that monitors the vital signals of firefighters responding to a fire incident and the temperature around them. The goal of the monitoring is twofold: (a) pull a firefighter out of the scene when a life-threatening condition is detected, and (b) make better decisions through fast and accurate assessment

of the situation. Each firefighter is monitored by a single wearable sensor, and therefore previously proposed fault tolerance strategies that rely on the availability of a back-up device capturing the same input are not applicable. The wearables are interconnected using a wireless network.

In this application, the most time sensitive operation is the detection of abnormalities in the firefighter's vital signals. To reduce the latency and increase the survival chances of the firefighter, we need to perform this operation as close as possible to the source, i. e. the wearable sensor. Even if the wearable suffers from a transient disconnection, it is critical to keep processing the captured data offline, as otherwise we might miss a sign that the firefighter needs help. That way, when the wearable regains connectivity to another network node, it can transmit an alert without delay.

To assess the overall situation, the fire area is divided in zones and every wearable generates a periodic report of the maximum temperature that was recorded in each zone. These reports are then aggregated, to obtain the global maximum temperature for each zone. This task uses the combined processing capabilities of multiple nodes in the network. To get an accurate assessment of the situation, we need to minimize the amount of data loss when wearables suffer from a transient disconnection. Furthermore, as the wearable sensors that generate data are worn by firefighters, they will move through the network with them. We therefore need to ensure that query execution resumes once connection is regained, even when a sensor moves to a different point in the network topology during the disconnection.

2 Background: NebulaStream

In the following, we give an overview of data stream processing, focusing on NebulaStream, the IoT data streaming platform into which we integrate our fault tolerance strategy.

IoT infrastructures are highly distributed, with sensors spanning entire cities or even countries. In addition, many IoT applications require prompt responses. One natural solution to support such applications are Stream Processing Systems (SPS), i. e. systems capable of performing computational tasks over a potentially infinite sequence of data items, called *tuples*. NebulaStream (NES) is a novel end-to-end SPS specifically designed for IoT applications [Ze20a, Ze20b], born from the need for a SPS that can better deal with the main challenges in IoT settings. NES aims to provide similar functionalities as widely used SPS such as Flink [CEH15] or Spark [Za16] while better supporting IoT applications. To that end, NES introduces mechanisms to handle, among others, the hardware heterogeneity in IoT environments, the high distribution of data and compute, and the unreliable communication in fog and sensor networks.

Prior systems employ either the cloud or the fog paradigm. Cloud-centric systems (e. g. Flink, Spark or Kafka [NSP17]) collect all data in a data center before processing them. Given that upcoming IoT applications will require processing data from millions of distributed sensors,

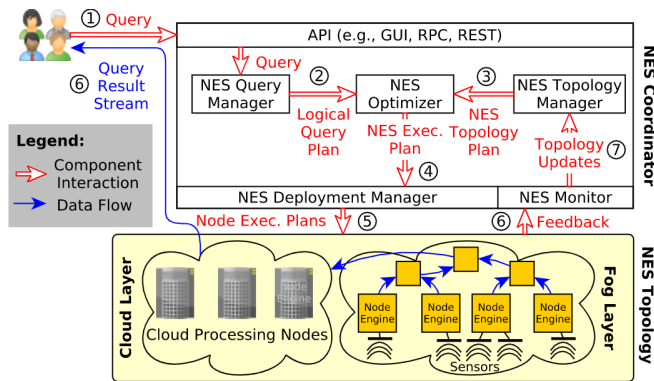


Fig. 1: NebulaStream's Architecture. Figure from [Ze20a].

this centralized approach results in a bottleneck and lacks scalability. Existing systems based on fog computing are also not optimal for the IoT. Systems like Frontier [OSP18] or CSA [Sh16] bring computations closer to the sources, thus reducing the bottleneck of cloud systems. However, they advocate a completely decentralized approach and only exploit the computing capabilities of nodes at the edge of the network. In the field of Wireless Sensor Networks (WSN) we can find Database Management Systems (DBMS) such as TinyDB [Ma05] that exploit the combined capabilities of sensors and actuators. These systems provide strategies to optimize query execution for devices with limited battery life but offer neither strong fault tolerance nor correctness guarantees. NES takes the state-of-the-art one step further and unifies the cloud, fog and sensor layers, leveraging the individual advantages of each layer and enabling optimizations across them. Specifically, data are typically generated within the sensor layer and routed through intermediate nodes up to the cloud. Any of the nodes in the path between the sensors and the cloud can access any data being routed through them and perform data processing tasks [Ga20]. The cloud performs any remaining processing as a fall-back.

NES Topology. There are three types of nodes in NES: *Workers*, *Sensors*, and a *Coordinator*. *Workers* process data by employing *operators* that consume tuples, apply a computational task (e.g. filter, map, sum, count), and emit new tuples. *Sensors* generate data and can also perform computational tasks. Mobile devices in NES fall in that category. Finally, the *NES Coordinator* is in charge of administering the network: it is aware of the current topology, it registers/deregisters nodes and streams, and deploys/undeploys queries. A single physical device can potentially host multiple NES nodes.

The network topology maintained in the coordinator is modeled as a graph consisting of sensor, worker and coordinator nodes and network links among them. Initially, the network is formed solely by the coordinator and one worker contained in it. Further workers and sensors can join the network by sending a request to the coordinator. They are then added to the topology graph. In the case of sensors, the streams that the sensors generate are also

registered in the topology graph. Once registration is complete, the new node is ready to start participating in query processing. When a node needs to exit the network, it does so by sending a deregistration request to the coordinator. The coordinator then removes the node and any streams it might generate from the topology graph, and undeploys the affected queries that were using the removed streams.

NES Query Deployment. Figure 1 displays an overview of NebulaStream’s architecture. A query submitted through the NES API ① contains information regarding the streams from which data should be obtained and the computational tasks that should be applied to them. Currently, NES uses a centralized deployment process. The coordinator is comprised of several components that handle different aspects of the deployment. First, the *NES Query Manager* transforms the user query into a directed acyclic graph (DAG) that contains source, processing, and sink nodes and directed links among them. A *source* is a node that generates data streams, i. e. provides the input, and a *sink* is a node that consumes data streams without generating new ones, i. e. provides the output. The links represent the communication channels for the data exchange among operators. The DAG is essentially a logical query plan that describes conceptually the operations that need to be performed over a data stream. To execute queries in a SPS, we need to generate a physical query execution plan (QEP) which maps the logical query plan to physical nodes. The *NES Optimizer* takes the query DAG from the Query Manager ② and the topology graph from the *NES Topology Manager* ③ and creates a physical query execution plan (QEP) which dictates the assignment of the different operators in the logical query plan to physical nodes in the topology ④. The process of assigning operators to physical nodes is known as *operator placement*. The next paragraph discusses operator placement in further detail. The *NES Deployment Manager* takes the execution plan and deploys the operators to their assigned nodes in the topology ⑤. Finally, the *NES Monitor* collects information about the changes occurring in the network topology ⑥ and sends the updates to the Topology Manager ⑦.

NES Operator Placement. The choice of the placement strategy depends on the optimization goal, e. g. low latency, high throughput, or minimal use of resources. NES supports several operator placement strategies. In addition to them, we also implemented a shortest-path based placement strategy. In contrast to the already existing strategies that always place the sink operator on the coordinator, the shortest-path based placement strategy allows to place the sink operator on any arbitrary node in the network. That way, we enable greater proximity to end-users and can avoid streaming data to the cloud-based coordinator. Clearly, the source operator is assigned to the sensor node that generates the data for the queried stream. The shortest-path based strategy places the remaining operators as follows. It first finds the shortest path between the source and the sink. Then, it applies a “push-down as far as possible” strategy: It starts placing operators on the shortest path from the source all the way up to the sink according to the available resources on each node, i. e. while there are remaining resources on a node, it keeps assigning operators to it. This strategy is particularly suited for queries that consist of filter operators as it reduces downstream data traffic. Once the assignment is complete, the coordinator sends to each of the nodes the part of the query execution plan that they need to execute (i. e. a pipeline of operators), and

indicates from which node they will receive their input, and to which node they should send their output. Some nodes in the path might be assigned a forward operator, which means that their only mission is to transmit data to the next node in the execution plan without processing them. Our fault tolerance solution is independent of the choice of the placement strategy and can thus be used in conjunction with any strategy. The NES Optimizer uses a placement strategy to generate a QEP given a query DAG and a network topology. As we will describe in more details in Section 3, our approach does not modify the QEP. It simply determines *when* a new QEP needs to be generated upon a device re-connection, and analyzes the newly generated QEP to determine *how to forward* the data that have been buffered during a data source disconnection.

3 Fault Tolerance for Mobile Data Sources

Our approach, NebulaStream-MSS, is an extension of NebulaStream that provides fault tolerance capabilities for mobile data sources such as mobile phones and wearables. Our approach is holistic: it handles the disconnection process, the data buffering during the disconnection, as well as the process of resuming query execution after the re-connection. Specifically, our approach employs a circular buffer that temporarily stores the data that are processed locally during a transient disconnection. We couple the buffer with a query-aware replacement policy that evicts data when the buffer becomes full before connectivity is regained without penalizing any query unequally. To handle the mobility of devices during a disconnection, we combine buffering with a restart process that determines which of the queries that involve the device require redeployment. When a query is redeployed, the device might be assigned different tasks than before the disconnection. Our restart process handles this situation by generating new paths through which the buffered data can be forwarded directly to the appropriate node, ensuring that the buffered data are processed correctly. We describe NebulaStream-MSS in more details in the following sections.

3.1 Disconnection Process

Our solution aims to handle transient disconnections of mobile devices. This raises the question of how we can distinguish a transient disconnection from a more permanent failure. From the point of view of the coordinator both look the same: a node has unexpectedly disappeared from the network. The short answer is that we cannot know with certainty when a disconnection is transient. We, however, follow an optimistic approach where at first we assume that all the disconnections that happen to mobile devices are transient, i.e. once a mobile device disappears from the network, the coordinator assumes that it will return. This is in contrast to the current approach in NES, where the coordinator completely removes the node and any associated streams or queries from the system upon a disconnection. If the device does not return after a timeout period, the coordinator assumes that the disconnection is permanent and acts accordingly. Choosing the timeout duration is challenging and depends on both the network characteristics and the application requirements, as different applications can tolerate different delays. The problem of finding the timeout duration is orthogonal to the problem addressed in this work and we leave it open for future research.

3.2 Data Buffering

Mobile devices acting as data sources can still both generate data and perform computations while transiently disconnected. Furthermore, in many real-world scenarios (e. g. the one described in Section 1.1), we have no control over the data source, and thus pausing data generation or modifying the data generation rate is infeasible. Therefore, to avoid data loss during a disconnection, we need to temporarily store the generated data on the device. To reduce the amount of data that needs to be stored and to exploit the offline processing capabilities of the device, we also propose to keep processing the data locally during the idle disconnection period. To that end, we base our solution on a buffering component that stores the data that are generated and processed on a data source node during a disconnection period. In the following, we discuss our buffering component in more details.

Buffer Structure. To ensure efficient space utilization and minimize access latency, we use a circular buffer. Circular buffers are particularly suited when the data are accessed in a First In First Out (FIFO) order. Our buffer stores batches of tuples after they are processed. As our device may have several streams or queries attached to it, every batch may have a different size and contain tuples of different formats. To retrieve batches from the buffer, we need to be able to infer their size. For this reason, we always precede batches with a control block, i.e. a small memory region in the buffer that contains metadata about the batch of tuples that follows it. Specifically, a control block stores a query identifier that indicates the query to which the batch belongs, the tuple size, and the number of tuples in the batch. Figure 2 presents the conceptual view of our circular buffer. As with any circular buffer, we always maintain two pointers: one pointing to the address of the first control block stored in the buffer (*Read Pointer*) and one pointing to the first free address in the buffer where new batches can be added (*Write Pointer*).

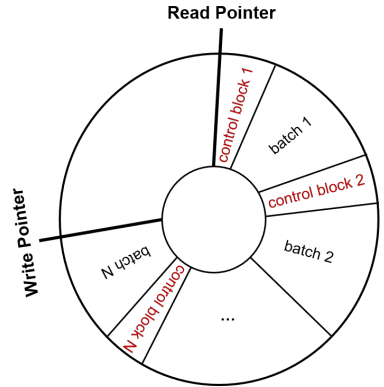


Fig. 2: Conceptual view of the circular buffer in NebulaStream-MSS.

Data Insertion. Once the data source has executed all operators assigned to it over the generated data, it sends the processed batch to the buffer. Inserting the batch into the buffer is fairly trivial as long as there is enough available space to store the incoming batch and the corresponding control block. The control block is simply written into the first free address in the buffer indicated by the Write Pointer, followed by the batch of tuples.

Query-aware Replacement Policy. As stated before, our buffer might store results corresponding to different queries. To achieve fairness among different queries running on the device, we propose a query-aware FIFO replacement policy that aims to always keep some results of each query in the buffer. Specifically, whenever we need to make space for a new batch of tuples, our policy replaces the oldest batch of the same query if possible.

If there are no other batches of the same query, or if removing the batch is insufficient, it removes the oldest batch of the query with the highest number of batches in the buffer. The process is repeated until there is enough space in the buffer for the new batch. One drawback of this policy is that it might cause segmentation in the buffer, i.e. fragments of empty space. To address this, whenever we remove a batch from the buffer, we shift all subsequent batches up, filling up the generated gap. Algorithm 1 presents the data insertion process using our query-aware FIFO replacement policy.

Algorithm 1: Data insertion with query-aware FIFO

```

1 Incoming Data: a batch of tuples batch for query with queryID
2 while (available space < (controlBlock.size + batch.size)) do
3   delID = queryID
4   if (there are no results of query with queryID in the buffer) then
5     | delID = query with highest batchCounter
6   end
7   delete oldest batch of query with delID
8   move up subsequent batches
9   batchCounter[delID] - -
10 end
11 copy batch at writePointer
12 writePointer += (controlBlock.size + batch.size)
13 batchCounter[queryID]++

```

Data Extraction. Once the mobile device regains connection to the network, we can proceed to send the buffered data. Extracting data from our buffer is fairly simple. Starting at the Read Pointer, we calculate up to what address in the buffer each result is stored, as we have information about the size of the batch in the control block. Since we use a circular buffer, no data shuffling is required on data extraction. Once we have retrieved a batch, we advance the Read Pointer up to the next control block.

3.3 Query Restart Process

Once the connection of a mobile device is reestablished, we have to restore the system to its normal working state. First, the device notifies the coordinator of its return, indicates the queries it was involved in, and transmits its current position by announcing which devices are in its neighborhood. Based on that information, there are two possible scenarios that determine the actions that should be taken by our system: re-connection without mobility and re-connection with mobility. In the first scenario, the device can still reach its sink nodes. This typically happens when the device did not move, or moved very little, during the disconnection period. In the second scenario, the device can no longer reach its original sink nodes, which happens in the case of high mobility.

Re-connection without Mobility. In this scenario, query execution can continue as normal, i.e. there is no need to modify the query execution plan. Once the coordinator receives the re-connection notification, it checks the position of the device and determines

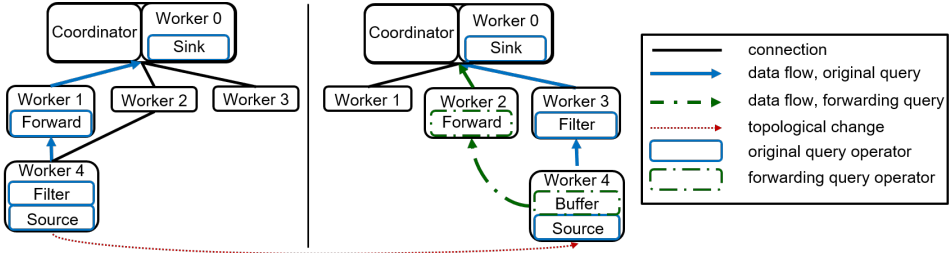


Fig. 3: Mobility during the execution of a simple filtering query. Before the disconnection (left), the device (Worker 4) acts as the source in the QEP, and also applies a filter operator. During the disconnection, Worker 4 moves out of the range of its downstream node (Worker 1). Once it returns at a new point in the network (right), the new QEP (shown in blue) assigns the filter operator to Worker 3. As the buffered data have been already filtered, we want them to bypass Worker’s 3 filter. This is achieved by deploying a forwarding query (shown in green) that sends the buffered tuples to Worker 2.

that the device remains connected to its previously assigned sink nodes. Then, query execution resumes and the device transmits the data stored in the buffer.

Re-connection with Mobility. In this scenario, the device moves through the network during the disconnection period, reappearing at a different position in the topology than the one that it last registered for. When this occurs, the mobile device might be unable to reach the sink nodes that had been previously assigned to it. Figure 3 shows an example: the mobile source (Worker 4) was original connected to Workers 1 and 2 (left). After the disconnection (right), it remains connected to Worker 2, but moves out of Worker’s 1 range. Instead, it is now connected to Worker 3. This mobility affects query execution and requires action from the system. Once the coordinator receives the notification that the device has returned, it can see from its current position that it has moved. As a result, it triggers the redeployment of every query for which the device can no longer connect to its sink nodes. The coordinator can employ any of the available operator placement algorithms to perform the redeployment. There are three possible scenarios that can arise in the newly generated query execution plan: the device is assigned (a) the **same** operators as before, (b) a **superset** of the original operators, or (c) a **subset** of the original operators.

If the data source is expected to apply the same set of operators over new data once the re-connection is complete, then all operators have already been applied to the data that were processed during the disconnection period. Therefore, we can simply retrieve the data from the buffer and send them to the nodes indicated by the new query execution plan.

If the data source is assigned a superset of the operators that it was previously executing, the data contained in the buffer have not been processed fully. In this case, every time a batch of data is retrieved from the buffer, we first need to apply the remaining operators in the pipeline to it, before sending it to the nodes specified in the query execution plan.

In the last scenario, the data source is assigned a subset of the original operators. In this case,

we cannot simply send the buffered data to the next node in the new query execution plan, as that node would re-apply some operators that have already been applied to them. For that purpose, we propose a mechanism that forwards the buffered data to the pertinent node in the network and skips over the operators that have already been applied to them. As described before, NES supports the forwarding of data through the topology using forward operators. Therefore, a natural way to create a forwarding path is by issuing a new query that uses forward operators to bypass the operators which have already been applied to the buffered data. We call this query a *forwarding query*. The coordinator generates automatically the forwarding query as follows. First, it specifies that the source of the forwarding query is only the buffered data, and not the newly generated tuples. Then, it examines the QEP of the original query to determine the node that contains the first operator that has not been applied to the buffered data yet. This node is the sink of the forwarding query, named *fwd_sink*. Next, the coordinator executes the shortest-path based operator placement algorithm and places a forwarding operator in every node of the path between the source and the *fwd_sink*. Lastly, the coordinator has to determine the operators that will be placed on the *fwd_sink* node. To do so, it examines the operators that are assigned to the *fwd_sink* node in the original QEP. If none of the operators has been applied to the buffered data, then the coordinator does not need to place any additional operators on the *fwd_sink* node. Instead, the tuples of the forwarding query will join the QEP of the original query. This is achieved by notifying the *fwd_sink* node about the mapping between the original and the forwarding query, so that the sink node can match the information arriving from the forwarding query to the original QEP. However, if the *fwd_sink* node contains some operators that have already been applied to the buffered data, the forwarding and the original query cannot be merged yet. Instead, the coordinator generates a new pipeline of operators for the forwarding query that contains only the operators that have not been applied to the buffered data yet. In this case, the tuples of the forwarding query join the original QEP in the next downstream node. Figure 3 shows a simple example of mobility during disconnection that requires the deployment of a forwarding query. Forwarding queries are short-running: once the buffer is empty, they are removed from the system and processing continues normally.

Imagine now that for the scenario shown in Figure 3 (right), one of the nodes connecting the source and sink in the forwarding query suffers from a disconnection as well. To handle such cascading disconnections, we enable a certain level of replication in our forwarding queries. We do so by finding as many paths between the source and the sink node as possible, up to a maximum indicated by a user-specified replication rate. The buffered data are then redundantly transmitted through all those paths, to avoid data loss in case of an intermediate node suffering from a failure. Each path is modelled as a separate forwarding query, i.e. we issue multiple forwarding queries concurrently that all have as input the same buffered data. To avoid processing the same data received through different paths multiple times at the sink node, each batch is assigned a timestamp and the sink node keeps track of the timestamps that it has already seen for each query. These timestamps are flushed periodically at the sink node, as we no longer expect to see duplicate batches after a certain period of time.

4 Experimental Evaluation

In this section, we first describe the experimental setup and then present the evaluation of our approach to support mobile sources in NebulaStream. The aim of the performed experiments is to showcase the fault tolerance capabilities of our approach and its efficiency when compared with the vanilla NebulaStream. Section 4.2 presents a comparative analysis while section 4.3 analyzes the performance of our approach in more details.

4.1 Experimental Setup

Hardware. The experiments were performed on a machine with a 10th generation Intel® Core™ i7 processor and 16 GB of RAM running Ubuntu 20.04 LTS.

Implementation. NES (and therefore also NebulaStream-MSS) is implemented in C++. Control messages between the coordinator and other nodes are handled through gRPC [GRP], while data transfer is performed using ZeroMQ [Hi].

Data. As we explain in Section 1, in our work we assume that we cannot control the data source, i. e. we cannot pause and restart the source and we cannot modify the data generation rate. The data source simply generates data infinitely, at a certain frequency. However, in our experiments we want to show the benefit of processing data during the disconnection period, which is what our approach enables, versus pausing query execution completely during the disconnection, which is what NebulaStream does. For that purpose, we use a CSV file as a source, which allows to pause data generation and easily restart it by storing the last read position in the file. We use the DEBS 2013 Grand Challenge dataset (described in [MZJ13]) which contains data generated from sensors attached on football players during a match. Since we focus on a single data source, we extract the data corresponding to a single player, the one with ID 10. The resulting CSV file contains a total of 6,575,830 tuples, each of 62 bytes. In all the experiments, our CSV data source produces data at a rate of 3.33 MBps, i. e. we read 56,375 tuples every second. In addition, we use a binary generator that produces tuples containing a single 64-bit unsigned integer field at a rate of 488 KBps.

Queries. We use two queries, one for each of the aforementioned data sources. The first, referred to as “Query 1”, is a filtering query that takes as input the CSV source. It applies two filters: filter 1 selects tuples that satisfy the condition $vx > 0$ and filter 2 selects tuples satisfying the condition $az > 1$. The overall query selectivity is 24%. Once filtered, the results are written to a file. Our second query, “Query 2”, simply streams the tuples of the binary generator source through the topology and prints them in standard output at the sink.

Network Topology. All the experiments presented in this evaluation, start with the same network configuration depicted in Figure 4 (left), consisting of the coordinator (that also contains the sink), and 8 other workers, one of which contains the source for both queries. Even though in a real-world scenario the full topology of the network would be significantly larger, we consider our topology to be representative of the relevant part of the network

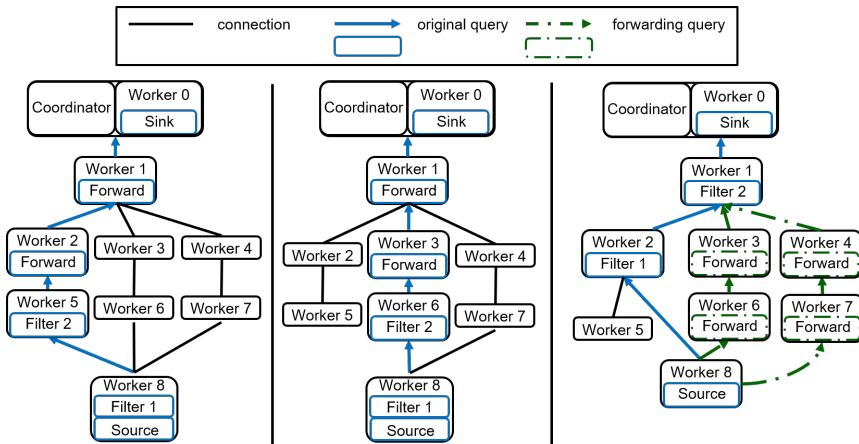


Fig. 4: Three topology snapshots along with the placement of Query 1 operators. (Left): Initial topology and placement. (Middle): After a transient disconnection, worker 8 gets disconnected from worker 5, which affects the placement of Query 1. There is no need for a forwarding query, as worker 8 applies the same operator as before (filter 1). (Right): After a transient disconnection, worker 8 gets disconnected from worker 5 and connected to worker 2. In the new placement, worker 8 no longer applies filter 1, and thus two forwarding queries are deployed.

that is involved in processing data captured by a given mobile data source. As we deal with mobility of the data source during disconnections, the topology is not static. Following a transient disconnection, the source might re-connect at a different location within the network. The middle and right parts of Figure 4 show the different topology snapshots that we use to simulate mobility. Given the different topology snapshots, we deploy Query 1 as described next. As you can see in Figure 4 (left), in the initial topology, one filter operator is placed at the source (worker 8), worker 5 applies the second filter, while workers 2 and 1 forward the data to the sink. The placement shown in Figure 4 (middle) is similar to the original one. The only difference is that, since now the source is disconnected from worker 5, the second filter is applied by worker 6, while it is worker 3 that forwards the results to worker 1. The last operator placement corresponds to the re-connection scenario that requires query redeployment, and the source is assigned a subset of the original operators after the redeployment (Figure 4 (right)). In this case, filter 1 is already applied to the buffered data, so we forward them directly to worker 1 which applies the second filter. For that, we employ the two forwarding queries shown with green color. “Query 2” (not shown in the figure), simply forwards the generated data through the shortest path.

Configuration Parameters. Unless otherwise stated, the buffer size is set to 50MB. We chose this size considering the characteristics of modern smartphones and wearables, but also the fact that we focus on short disconnections during which a limited amount of data are generated. When applying data forwarding, we use a replication factor of 2.

Evaluation Metrics. We use two metrics throughout our evaluation: query runtime and data loss ratio. Query runtime gives us insight into the performance of our approach, while the data loss ratio indicates the level of fault tolerance that we can achieve. We define *query runtime* as the time between the moment that a new query starts processing data and the moment that the query has processed a fixed amount of data. Specifically, in our runtime experiments we measure the total time that it takes for Query 1 to process 2M tuples of our CSV data source. *Data loss ratio* is the ratio of data that were evicted from the buffer due to lack of space over the total amount of data that were generated during the disconnection period after applying all local operators. Specifically, the data loss ratio (%) is calculated over a fixed disconnection period as: $\frac{\text{amount of data evicted from the buffer}}{\text{total amount of generated data after applying local operators}} \cdot 100$. We ran each experiment 5 times and report the average results.

4.2 Comparative Analysis

We first perform a comparative analysis of NebulaStream-MSS with the vanilla implementation of NebulaStream. The baseline NES considers all node disconnections to be final, i. e. upon a disconnection, the node is completely removed from the system. In the case of a data source, this results in halting the execution of the queries that the source is involved in. Furthermore, the data that are produced during a disconnection are lost.

Data Loss Ratio. This experiment explores the fault tolerance capabilities of NebulaStream-MSS, based on the amount of data that are lost during transient disconnections. We assume that the mobile data source continues to produce data while being transiently disconnected from the other nodes in the network, as it would happen with a sensor in a real-world scenario. We run concurrently Query 1 and Query 2 in the topology of Figure 4 (left), with worker 8 hosting the data sources for both queries. Worker 8 also applies Filter 1 of Query 1, even during the disconnection, and only buffers the filtered data. The filter selectivity is 67.6%. To show the benefits of our query-aware FIFO replacement strategy, we also implemented a simple FIFO strategy. As Figure 5 shows, the replacement strategy impacts the amount of data loss. This is because we are running two different queries that generate batches of different sizes, 3.33 MB and 488 KB respectively. Since the FIFO strategy always removes the oldest batch, even when we only need space for 488 KB in the buffer, we might end up removing a batch of 3.33 MB, if this batch is the oldest. This

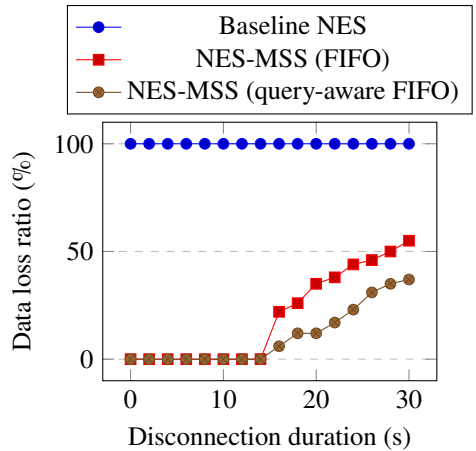


Fig. 5: Data loss ratio during a transient disconnection. After 15 seconds of disconnection, the buffer becomes full and tuples start to be dropped.

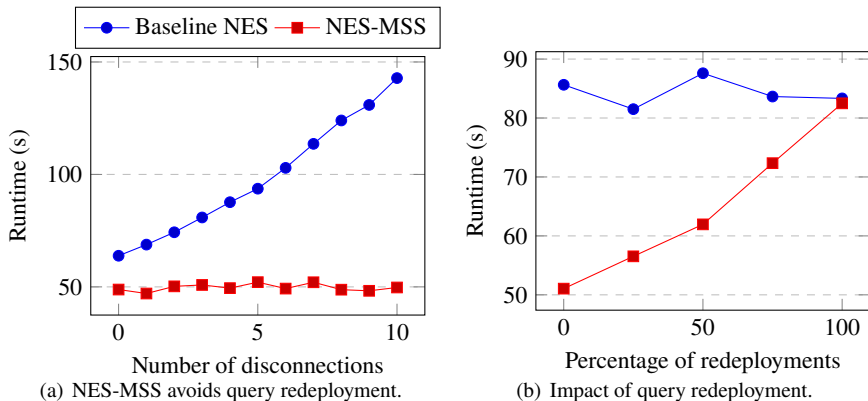


Fig. 6: Runtime of Query 1 (processing 2M tuples). 6(a): By avoiding query redeployment, NES-MSS has a constant runtime irrespective of the number of disconnections. 6(b): NES-MSS's performance converges to the one of baseline NES when the query needs to be redeployed after every disconnection.

can rapidly escalate the amount of data loss. Our query-aware FIFO strategy is far less susceptible to the time at which batches arrive at the buffer since we remove batches of the same query to make space for new data.

Query Runtime. We evaluate the runtime of Query 1 (i. e. the total time for processing 2M tuples), with a varying number of disconnections occurring during its execution. In order to be able to fairly compare the performance of our approach (that buffers data) with baseline NES (that loses any data generated during a disconnection), in this experiment we use a source that can be paused and restarted so that both systems process the same data throughout the experiment. Figure 6(a) shows the case where the query does not need to be redeployed after the disconnections. As can be seen in the figure, the runtime of baseline NES increases with the number of restarts while the runtime of our approach is almost constant. This is because baseline NES views the failure as permanent and removes the disconnected data source and the queries running on it from the system. When the source regains connection, it is treated as a new node: first, the coordinator has to re-add the source to the topology, and to register the streams that it generates in the system. Then, the queries applied on the generated streams are redeployed and restarted (as described in Section 2). Our approach, on the other side, does not remove the source and the queries from the system, keeps processing data locally at the disconnected source, and resumes execution fast once the source re-connects. In a real-world scenario, we expect to see multiple transient disconnections of a given device over time, due to mobility and network instability. It is likely that in some cases query redeployment will be required. We therefore study the impact of query redeployment on execution and show the results in Figure 6(b). For a fixed number of disconnections occurring during the processing of 2M tuples with Query 1, we evaluate how the runtime varies based on the percentage of disconnections that require redeployment. As expected, we found that with fewer redeployments, data are processed faster. On the other side, when we need to redeploy queries after every single disconnection, the performance

of NebulaStream-MSS converges to the one of baseline NES. That indicates the importance of avoiding query redeployment whenever possible and having a lightweight query restart mechanism, which is what our approach offers.

4.3 NebulaStream-MSS Analysis

In this section we take a closer look into the performance of our system to identify optimization opportunities.

Buffer Sensitivity Analysis. Figure 7 shows the impact of the buffer size on the data loss ratio for both buffer replacement strategies. We use the same setup as in the “Data Loss Ratio” experiment and we fix the disconnection duration to 30 seconds. As expected, there is an obvious correlation between the buffer size and the amount of the incurred data loss. Choosing an appropriate buffer size is vital to minimize the amount of data that are discarded during disconnections. The optimal size depends on the data generation rate, the disconnection duration, the query, and the available resources on the device. Furthermore, comparing the two strategies, we see that our query-aware FIFO strategy benefits more from a larger buffer size. This is because it makes better use of the available buffer space by dropping batches of the same size as the ones it needs to make space for. For 100 MB there is only a 6% difference between the two strategies, as the buffer can fit almost all the data that are generated during 30 seconds, while for 125 MB no data are discarded.

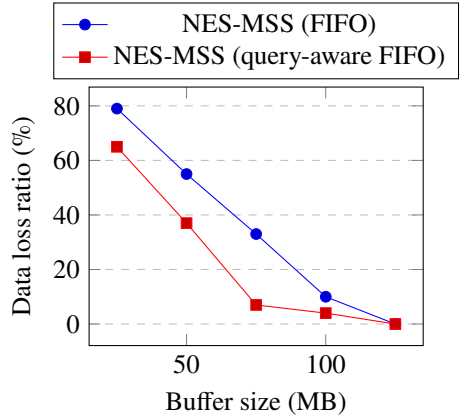


Fig. 7: Data loss ratio for a transient disconnection of 30 seconds. A buffer of 125 MB can fit all the generated data. When the available buffer space is more limited, the query-aware FIFO strategy can make better use of it.

Recovery Time Breakdown: Impact of Topology Size. This experiment investigates the time elapsed between the moment that a source executing a query regains its connection and the moment that the system resumes normal execution for different topology sizes. We run Query 1 on three topology configurations. *Topo5* corresponds to the topology of Figure 4 (left), where the query path between the source (worker 8) and the sink (worker 0) contains 5 nodes in total, while there are two more paths of size 5 between worker 8 and worker 0 (worker 8 -> worker 6 -> worker 3 -> worker 1 -> worker 0 and worker 8 -> worker 7 -> worker 4 -> worker 1 -> worker 0). *Topo4* is the same as *topo5*, but all paths are of size 4. Similarly, *topo7* is the same as *topo5*, but all paths are of size 7. The source disconnects after processing 2M tuples and re-connects after a short disconnection period. As Figure 8(a) shows, the largest portion of time is spent on redeploying the original query

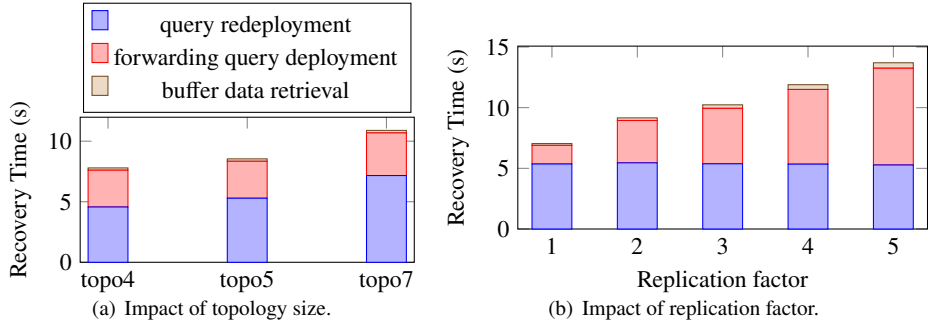


Fig. 8: Recovery time breakdown. 8(a): The redeployment time increases for larger topologies. 8(b): The time to deploy forwarding queries increases for higher replication factors.

and on creating and deploying forwarding queries, i. e. query deployment is clearly the main overhead. In addition, as expected, query redeployment takes longer for bigger topologies. We note, however, that we perform query redeployment only whenever necessary, i. e. when the source can no longer reach its original downstream node upon re-connection. In section 6 we discuss some possible directions to further mitigate the redeployment overhead.

Recovery Time Breakdown: Impact of Replication Factor. We repeat the previous experiment, but this time we vary the replication factor of the forwarding queries. The topology used in this experiment is similar to the one of Figure 4 (left) but wider: it contains 8 additional paths between worker 8 and worker 0. As expected, the results in Figure 8(b) show that the time to create and deploy forwarding queries increases for higher replication factors. The time to retrieve and send the buffered data also increases slightly, as the data are sent to more nodes. The query redeployment is independent of the forwarding queries, and remains, thus, constant. Overall, there is a trade-off between the level of fault tolerance and the recovery time, which we plan to further investigate in future work.

5 Related Work

Mobile Stream Processing. Mobile Stream Processing (MSP) [Ni15] refers to performing stream processing tasks purely at the edge by combining resources of multiple mobile devices. Given that our work focuses on the use of mobile devices in stream processing, fault tolerance strategies that have been proposed in the context of MSP are the closest to our work. Existing fault tolerance strategies in MSP fall broadly into two categories: checkpointing-based and replication-based. MobiStreams [WP14] combines two checkpointing protocols, token-triggered and broadcast-based checkpointing, aiming to reduce the network overhead that checkpointing strategies typically induce. In token-triggered checkpointing, a controller node periodically prompts data sources to checkpoint their state. To coordinate the checkpointing process, the data sources generate tokens that are sent to the downstream node upon completion of the checkpoint. Once a node receives tokens of all its upstream neighbors, it

checkpoints its own state and sends its token downstream. To avoid losing the checkpointed state when nodes exit the network, the broadcast-based checkpointing protocol broadcasts nodes' states in the network. The main shortcoming of MobiStreams is that it does not consider the limited resources of mobile devices. Checkpointing operations and state broadcasting have a considerable impact on battery life. In addition, MobiStreams consumes memory resources on the devices to store the checkpointed state even when the system is functioning correctly, introducing an unnecessary overhead. In contrast, NebulaStream-MSS follows a reactive approach and only buffers data upon a disconnection. Symbiosis [MRH14] attempts to optimize checkpoints not only for network overhead, but also for energy efficiency. It proposes to trigger checkpointing when a node in the network either moves outside the range of connectivity, or reaches a critical battery threshold. However, Symbiosis only considers failures caused by mobility and energy levels and does not account for sudden ad-hoc disconnections caused by network instabilities.

On the other end of the spectrum we find systems that base their fault tolerance strategy on replication. Frontier [OSP18] models queries as replicated data flow graphs, where each operator is replicated in multiple nodes resulting in multiple paths between a source and a sink node. It then adjusts the data flow dynamically based on a backpressure stream routing algorithm. To recover from the disconnections of data sources, Frontier applies replication also at the source level. In IoT scenarios, however, it is not always possible to have multiple sources (i. e. sensors) for the same data. MobileStorm [Ni15], is a stream processing platform for clouds of mobile devices, but does not provide support for transient disconnections. To address that, R-MStorm [CS20], an extension of MobileStorm, follows a similar approach as Frontier. It introduces path diversity by replicating operators, so that there is always a path between the source and the sink, even in case of failures, and performs dynamic path selection. In addition, R-MStorm attempts to improve the overall system availability by using an operator placement strategy that prioritizes devices with higher availability during operator assignment. R-MStorm does not provide fault tolerance at the data source level and therefore cannot recover from transient disconnections of data sources. Finally, Swing [FSL18] is a MSP that considers the dynamism of mobile devices. Each upstream node in Swing maintains routing information about the reachable downstream nodes. When a network link is broken, the affected upstream nodes update their routing records and re-route data to other nodes. However, any data that are generated from the moment of the disconnection until the completion of the reconfiguration are lost. In contrast, we reduce or completely eliminate data loss by buffering data on the disconnected device.

Distributed Stream Processing. Existing distributed stream processing systems provide fault tolerance through one of the two following approaches [Hw05]: upstream backup, where nodes buffer sent data while the downstream nodes process them and replay them to a recovery node upon a failure [Qi13, STO], or replication, where each node is assigned a second node as a backup [Ba08, SHB04]. As we already explained before, replication-based approaches are not practical in IoT environments, given the limited amount of resources, and can only provide fault tolerance at the data source level when there are multiple devices

capturing the same input data. Similarly to our work, the upstream backup approach also relies on buffering. However, unlike our approach, upstream backups aim to handle failures at the receiver by buffering data at the sender. In our approach, we deal with failures at the data source which *does not have an upstream node*, i. e. is at the bottom of the QEP. We therefore need to buffer data at the source itself. Furthermore, while the upstream backup approach buffers data throughout query execution, our approach only buffers data reactively upon a disconnection. Finally, our approach also considers topological changes when forwarding the buffered data.

Mobile Computing and Networking. In the past decades, there has been a lot of research in the area of data management in mobile computing [Ba99]. In [BI94], the authors propose different caching strategies for mobile devices. They categorize devices into sleepers and workaholics based on the duration of their disconnection and show the impact of the disconnection duration on the effectiveness of the caching strategy. Wu et al. [WYC96], address the problem of selectively discarding caches in mobile devices that have been disconnected for a period of time. In contrast to the above work, we assume that the contents of the buffer do not become obsolete during the disconnection period, as we are focusing on transient disconnections. Another line of work, aims to provide network connection mobility. Persistent Connections [YS95] interpose a library between the application and the sockets API that provides the illusion of a single unbroken connection over successive physical connection instances. When a physical connection is lost, the sender stores data in a buffer. Once a new physical connection is established, any data buffered during disconnection are sent through it. Similarly, our approach uses buffering at the data source during a disconnection. However, instead of using an external library, we tightly integrate buffering inside the streaming engine. Furthermore, unlike Persistent Connections, our approach is holistic: in addition to buffering, we provide a query restart process that resumes query execution upon a re-connection.

Buffering for Fault Tolerance. Clearly, there are plenty of fault tolerance strategies in different domains that rely on buffering. Message logging and checkpointing are often used for fault tolerance in distributed systems [EI02]. In [ZJ87], the sender stores messages in its local memory, which allows recovery from single failures, while [SBY88] builds upon that approach by selectively logging only data that cannot be otherwise reconstructed. Unlike our work, the above techniques do not support mobility of the sender. In addition, they follow a *pessimistic* approach that logs messages irrespective of failures, while we only buffer data upon a disconnection. DiscoTech [RGG12] is a toolkit for handling disconnections in groupware networks. It includes fault tolerance strategies that use event queues to store data in a centralized server during the disconnection of a receiver node. In contrast, we deal with failures at the source that sends the data, and store the data locally during the disconnection.

6 Conclusions & Future Work

This paper presents NebulaStream-MSS, an extension of NebulaStream that provides fault tolerance capabilities for mobile sources in IoT environments. NebulaStream-MSS focuses specifically on overcoming transient disconnections. At the core, the proposed solution is a data logging mechanism that allows mobile devices to continue processing data while they are in a disconnected state by temporarily storing the processed data in a circular buffer. That way, we exploit the processing capabilities of the mobile devices and transform the idle disconnection time into a productive period. Besides buffering data during transient disconnections, our fault tolerance strategy also includes a query restart process that ensures the consistent resumption of query execution, even when mobile devices move during the period of disconnection. Our restart process determines whether queries need to be redeployed, based on whether the device can still reach its downstream neighbours after the re-connection. In addition, we introduce forwarding queries, to address the case where the mobile device is assigned a subset of the operators previously assigned to it after a new deployment. These forwarding queries create new paths in the DAG of a query through which the data stored in the buffer can be forwarded further down the pipeline to avoid redundant processing. Using a custom benchmark based on real data [MZJ13], we show that NebulaStream-MSS reduces data loss by 63% over a disconnection period of 30 seconds and provides nearly constant query runtime with an increasing number of disconnections when no query redeployment is required.

In future work, we plan to further improve our buffering strategy. Since mobile devices have limited resources, minimizing the memory requirements is critical. For that purpose, we would like to investigate the use of more tailored replacement strategies that exploit application knowledge to determine the most relevant information. In addition, we plan to look into data compression and result sharing. Result sharing would allow us to only once store duplicate results produced by different queries. Moreover, we plan to further investigate the selection of an optimal buffer size based on the available resources of the device and the workload. Our evaluation also showed that the largest performance overhead stems from the redeployment of queries. To mitigate this overhead, we aim to perform dynamic and partial redeployment. That way, we can reconfigure the data flow without involving the coordinator thereby eliminating our system's bottleneck. Finally, we have proposed the use of a level of replication in our forwarding queries. To reduce network traffic, instead of broadcasting the data through all redundant paths, we could use an approach similar to Frontier's [OSP18] where data are sent through a single path chosen dynamically at runtime.

Acknowledgements

This work was supported by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref 01IS18037A).

Bibliography

- [Ba99] Barbara, D.: Mobile computing and databases-a survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):108–117, 1999.
- [Ba08] Balazinska, Magdalena; Balakrishnan, Hari; Madden, Samuel R.; Stonebraker, Michael: Fault-Tolerance in the Borealis Distributed Stream Processing System. *ACM Trans. Database Syst.*, 33(1), March 2008.
- [BI94] Barbará, Daniel; Imieliński, Tomasz: Sleepers and Workaholics: Caching Strategies in Mobile Environments. In: *International Conference on Management of Data SIGMOD*. p. 1–12, 1994.
- [CEH15] Carbone, Paris; Ewen, Stephan; Haridi, Seif: Apache Flink: Stream and Batch Processing in a Single Engine. In: *IEEE Data Engineering Bulletin*. volume 36, 2015.
- [CS20] Chao, Mengyuan; Stoleru, Radu: A Resilient Mobile Stream Processing System for Dynamic Edge Networks. In: *IEEE International Conference on Fog Computing (ICFC)*. 2020.
- [EI02] Elnozahy, E. N.; Alvisi, Lorenzo; Wang, Yi-Min; Johnson, David B.: A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
- [FSL18] Fan, S.; Salonidis, T.; Lee, B.: Swing: Swarm Computing for Mobile Sensing. In: *International Conference on Distributed Computing Systems (ICDCS)*. pp. 1107–1117, 2018.
- [Ga20] Gavrilidis, Haralampos; Michalke, Adrian; Mons, Laura; Zeuch, Steffen; Markl, Volker: Scaling a Public Transport Monitoring System to Internet of Things Infrastructures. In: *International Conference on Extending Database Technology (EDBT)*. pp. 627–630, 2020.
- [GRP] Introduction to gRPC, <https://grpc.io/docs/what-is-grpc/introduction/> Last accessed 12/12/2020.
- [GS] The Mobile Economy 2020, https://www.gsma.com/mobileeconomy/wp-content/uploads/2020/03/GSMA_MobileEconomy2020_Global.pdf Last accessed at 23/09/2020.
- [Hi] ZeroMQ guide: Preface, <http://zguide.zeromq.org/page:preface> Last accessed 12/12/2020.
- [Hw05] Hwang, J. .; Balazinska, M.; Rasin, A.; Cetintemel, U.; Stonebraker, M.; Zdonik, S.: High-availability algorithms for distributed stream processing. In: *2IEEE International Conference on Data Engineering (ICDE)*. pp. 779–790, 2005.
- [Ma05] Madden, Samuel R.; Franklin, Michael J.; Hellerstein, Joseph M.; Hong, Wei: Tinydb: An acquisitional query processing system for sensor network. In: *ACM Transactions on Database Systems (TODS)*. 2005.
- [MRH14] Morales, Jefferson; Rosas, Erika; Hidalgo, Nicolas: Symbiosis: Sharing mobile resources for stream processing. In: *IEEE Symposium on Computers and communications (ISCC)*. pp. 1–6, 2014.
- [MZJ13] Mutschler, Christopher; Ziekow, Holger; Jerzak, Zbigniew: The DEBS 2013 Grand Challenge. In: *International Conference on Distributed Event-Based Systems (DEBS)*. p. 289–294, 2013.

- [Ni15] Ning, Qian; Chien-An; Stoleru, Radu; Chen, Congcong: Mobile Storm: Distributed Real-time Stream Processing for Mobile Clouds. In: IEEE International Conference on Cloud Networking (CloudNet). pp. 139–145, 2015.
- [NSP17] Narkhede, Neha; Shapira, Gwen; Palino, Todd: Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. O'Reilly, 2017.
- [OSP18] O'Keeffe, Dan; Salonidis, Theodoros; Pietzuch, Peter: Frontier: Resilient Edge Processing for the Internet of Things. In: PVLDB. volume 11, pp. 1178–1191, 2018.
- [Qi13] Qian, Zhengping; He, Yong; Su, Chunzhi; Wu, Zhuojie; Zhu, Hongyu; Zhang, Taizhi; Zhou, Lidong; Yu, Yuan; Zhang, Zheng: TimeStream: Reliable Stream Computation in the Cloud. In: European Conference on Computer Systems (EuroSys). p. 1–14, 2013.
- [RGG12] Roy, Banani; Graham, T.C. Nicholas; Gutwin, Carl: DiscoTech: a plug-in toolkit to improve handling of disconnection and reconnection in real-time groupware. In: ACM Conference on Computer Supported Cooperative Work. 2012.
- [SBY88] Storm, Robert E.; Bacon, David F.; Yemini, Shaula A.: Volatile logging in n-fault-tolerant distributed systems. In: International Symposium on Fault-Tolerant Computing. 1988.
- [Sh16] Shen, Zhitao; Kumaran, Vikram; Franklin, Michael J.; Krishnamurthy, Sailesh; Bhat, Amit; Kumar, Madhu; Lerche, Robert; Macpherson, Kim: CSA: Streaming Engine for Internet of Things. In: IEEE Data Engineering Bulletin. volume 38, 2016.
- [SHB04] Shah, Mehul A.; Hellerstein, Joseph M.; Brewer, Eric: Highly Available, Fault-Tolerant, Parallel Dataflows. In: International Conference on Management of Data SIGMOD. p. 827–838, 2004.
- [STO] Storm, <https://storm.apache.org/>.
- [WP14] Wang, Huayong; Peh, Li-Shiuan: Mobistreams: A reliable distributed stream processing system for mobile devices. In: IEEE International Parallel and Distributed Processing Symposium. pp. 51–60, 2014.
- [WYC96] Wu, Kun-Lung; Yu, Philip S.; Chen, Ming-Syan: Energy-Efficient Caching for Wireless Mobile Computing. In: International Conference on Data Engineering (ICDE). p. 336–343, 1996.
- [YS95] Yongguang Zhang; Son Dao: A “persistent connection” model for mobile and distributed systems. In: International Conference on Computer Communications and Networks (ICCCN). pp. 300–307, 1995.
- [Za16] Zaharia, Matei; Xin, Reynold S.; Patrick Wendell, Tathagata Das; Armbrust, Michael; Dave, Ankur; Meng, Xiangrui; Rosen, Josh; Venkataraman, Shivaram; Franklin, Michael J.; Ghodsi, Ali; Gonzalez, Joseph; Shenker, Scott; Stoica, Ion: Apache spark: a unified engine for big data processing. In: Communications of the ACM. 2016.
- [Ze20a] Zeuch, Steffen; Chaudhary, Ankit; Monte, Bonaventura Del; Gavrilidis, Haralampos; Giouroukis, Dimitrios; Grulich, Philipp M.; Breß, Sebastian; Traub, Jonas; Markl, Volker: The NebulaStream Platform: Data and Application Management for the Internet of Things. In: Conference on Innovative Data Systems Research (CIDR). 2020.

- [Ze20b] Zeuch, Steffen; Tziritza Zacharatou, Eleni; Zhang, Shuhao; Chatziliadis, Xenofon; Chaudhary, Ankit; Monte, Bonaventura Del; Giouroukis, Dimitrios; Grulich, Philipp M.; Ziehn, Ariane; Markl, Volker: NebulaStream: Complex Analytics Beyond the Cloud. *Open J. Internet Things*, 6(1):66–81, 2020.
- [ZJ87] Zwaenepoel, Willy; Johnson, D.B.: Sender-based message logging. In: *International Symposium on Fault-Tolerant Computing*. 1987.

Graph Sampling with Distributed In-Memory Dataflow Systems

Kevin Gomez^{1,2}, Matthias Täschner^{1,2}, M. Ali Rostami^{1,2}, Christopher Rost^{1,2},
Erhard Rahm^{1,2}

Abstract: Given a large graph, graph sampling determines a subgraph with similar characteristics for certain metrics of the original graph. The samples are much smaller thereby accelerating and simplifying the analysis and visualization of large graphs. We focus on the implementation of distributed graph sampling for Big Data frameworks and in-memory dataflow systems such as Apache Spark or Apache Flink and evaluate the scalability of the new implementations. The presented methods will be open source and be integrated into GRADOOP, a system for distributed graph analytics.

Keywords: Graph Analytics; Distributed Computing; Graph Sampling; Data Integration

1 Introduction

Sampling is used to determine a subset of a given dataset that retains certain properties but allows more efficient data analysis. For graph sampling it is necessary to retain not only general characteristics of the original data but also the structural information. Graph sampling is especially important for the efficient processing and analysis of large graphs such as social networks [LF06, Wa11]. Furthermore, sampling is often needed to allow the effective visualization and computation of global graph measures for such graphs.

Our contribution in this paper is to outline the distributed implementation of known graph sampling algorithms for improved scalability to large graphs as well as their evaluation. The sampling approaches are added as operators to the open-source distributed graph analysis platform GRADOOP³ [Ju16, Ju18] and used for interactive graph visualization. Like GRADOOP, our distributed sampling algorithms are based on the dataflow execution framework Apache Flink but the implementation would be very similar for Apache Spark. To evaluate horizontal scalability, speedup and absolute runtimes we use the synthetic graph data generator LDBC-SNB [Er15].

This work is structured as follows: We briefly discuss related work in Section 2 and provide background information on graph sampling in Section 3. In Section 4, we explain the distributed implementation of four sampling algorithms with Apache Flink. Section 5 describes the evaluation results before we conclude in Section 6.

¹ University of Leipzig, Database Group & ScaDS.AI Dresden/Leipzig

² [gomez, taeschner, rostami, rost, rahm]@informatik.uni-leipzig.de

³ <http://www.gradoop.com>

2 Related Work

Several previous publications address graph sampling algorithms but mostly without considering their distributed implementation. Hu et al. [HL13] surveyed different graph sampling algorithms and their evaluations. However, many of these algorithms cannot be applied to large graphs due to their complexity. Leskovec et al. [LF06] analyze sampling algorithms for large graphs but there is no discussion of distributed or parallel approaches. Wang et al. [Wal1] focuses on sampling algorithms for social networks but again without considering distributed approaches.

The only work about distributed graph sampling we are aware of is a recent paper by Zhang et al. [ZZL18] for implementations based on Apache Spark. In contrast to our work, they do not evaluate the speedup behavior for different cluster sizes and the scalability to different data volumes. Our study also includes a distributed implementation and evaluation of random walk sampling. Zhang et al. [Zh17] have evaluated the influence of different graph sampling algorithms on graph properties such as the distribution of vertex degrees as well as on the visualization of the samples. These aspects do not depend on whether the implementation is distributed and are thus not considered in this short paper.

3 Background

We first introduce two basic definition of a graph sample and a graph sample algorithm and then specify some selected sampling algorithms.

3.1 Graph Sampling

A directed graph $G = (V, E)$ can be used to represent relationships between entities, for example, interactions between users in a social network. The user can be denoted as a vertex $v \in V$ and a relationship between two users v and u can be denoted as a directed edge $e = (v, u) \in E$.

Since popular social networks such as Facebook and Twitter contain billions of users and trillions of relationships, the resulting graph is too big for both, visualization and analytical tasks. A common approach to reduce the size of the graph is to use graph sampling to scale down the information contained in the original graph.

Definition 1 (GRAPH SAMPLE) A graph $S = (V_S, E_S)$ is a sampled graph (or graph sample) that defines a subgraph of graph $G = (V, E)$ iff the following three constraints are met: $V_S \subseteq V$, $E_S \subseteq E$ and $E_S \subseteq \{(u, v) | u, v \in V_S\}$.⁴

⁴ In the existing publications, there are different approaches toward the vertices with zero-degrees in the sampled graph. Within this work we choose the approach to remove all zero-degree vertices from the sampled graph.

Definition 2 (GRAPH SAMPLE ALGORITHM) A graph sample algorithm is a function from a graph set \mathcal{G} to a set of sampled graphs \mathcal{S} , as $f : \mathcal{G} \rightarrow \mathcal{S}$ in which the set of vertices V and edges E will be reduced until a given threshold $s \in [0, 1]$ is reached. s is called *sample size* and is defined as the ratio of vertices $s_V = |V_s|/|V|$ (or edges $s_E = |E_s|/|E|$) the graph sample contains compared to the original graph.

3.2 Basic Graph Sampling Algorithms

Many graph sampling algorithms have already been investigated but we will limit ourselves to four basic approaches in this paper: *random vertex sampling*, *random edge sampling*, *neighborhood sampling*, and *random walk sampling*.

Random vertex sampling [Zh17] is the most straightforward sampling approach that uniformly samples the graph by selecting a subset of vertices and their corresponding edges based on the selected sample size s . For the distributed implementation in a shared-nothing approach, the information of the whole graph is not always available in every node. Therefore, we consider an estimation by selecting the vertices using s as a probability. This approach is also applied on the edges in the random edge sampling [Zh17].

We extend the idea of the the simple random vertex approach to improve topological locality using the random neighborhood sampling. Therefore, when a vertex is chosen to be in the resulting sampled graph, all neighbors are also added to the sampled graph. Optionally, only incoming or outgoing edges can be taken into account to select the neighbors of a vertex.

For the random walk sampling [Wa11], one or more vertices are randomly selected as start vertices. For each start vertex, we follow a randomly selected outgoing edge to its neighbor. If a vertex has no outgoing edges or if all edges were followed already, we jump to any other randomly chosen vertex in the graph and continue the walk there. To avoid keeping stuck in dense areas of the graph we added a probability to jump to another random vertex instead of following an outgoing edge. This process continues until a desired number of vertices have been visited, thus the sample size s has been met. All visited vertices and all edges whose source and target vertex was visited will be part of the graph sample result.

4 Implementation

The goals of the distributed implementation of graph sampling are to achieve fast execution and good scalability for large graphs with up to billions of vertices and edges. We therefore want to utilize the parallel processing capabilities of shared-nothing clusters and, specifically, distributed dataflow systems such as Apache Spark [Za12] and Apache Flink [Ca15]. In contrast to the older MapReduce approach, these frameworks offer a wider range of transformations and keep data in main memory between the execution of operations. Our implementations are based on Apache Flink but can be easily transferred to Apache Spark.

Transf.	Type	Signature	Constraints
Filter	unary	$I, O \subseteq A$	$O \subseteq I$
Map	unary	$I \subseteq A, O \subseteq B$	$ I = O $
Reduce	unary	$I, O \subseteq A \times B$	$ I \geq O \wedge O \leq A $
Join	binary	$O \subseteq I_1 \bowtie I_2$	$I_1 \subseteq A, I_2 \subseteq B$

(I/O : input/output datasets, A/B : domains)

Tab. 1: Selected transformations and their characteristics.

We first give a brief introduction to the programming concepts of the distributed dataflow model. We then outline the implementation of our sampling operators.

4.1 Distributed Dataflow Model

The processing of data that exceeds the computing power or storage of a single computer can be handled through the use of distributed dataflow systems. Therein the data is processed simultaneously on shared-nothing commodity cluster nodes. Although details vary for different frameworks, they are designed to implement parallel data-centric workflows, with datasets and primitive transformations as two fundamental programming abstractions. A *dataset* represents a typed collection partitioned over a cluster. A *transformation* is a deterministic operator that transforms the elements of one or two datasets into a new dataset. A typical distributed program consists of chained transformations that form a dataflow. A scheduler breaks each dataflow job into a directed acyclic execution graph, where the nodes are working threads and edges are input and output dependencies between them. Each thread can be executed concurrently on an associated dataset partition in the cluster without sharing memory.

Transformations can be distinguished into *unary* and *binary* operators, depending on the number of input datasets. Table 1 shows some common transformations from both types which are relevant for this work. The *filter* transformation evaluates a user-defined predicate function to each element of the input dataset. If the function evaluates to `true`, the element is part of the output. Another simple transformation is *map*. It applies a user-defined map function to each element of the input dataset which returns exactly one element to guarantee a one-to-one relation to the output dataset. A transformation processing a group instead of a single element as input is *reduce* where the input, as well as output, are key-value pairs. All elements inside a group share the same key. The transformation applies a user-defined function to each group of elements and aggregates them into a single output pair. A common binary transformation is *join*. It creates pairs of elements from two input datasets which have equal values on defined keys. A user-defined join function is applied for each pair that produces exactly one output element.

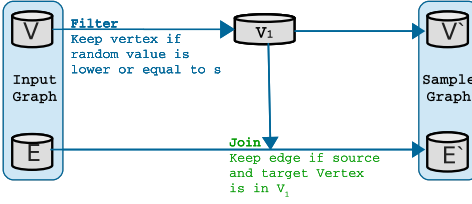


Fig. 1: Dataflow RV Operator.

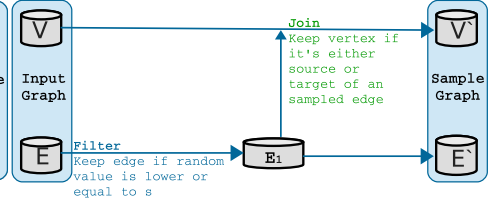


Fig. 2: Dataflow RE Operator.

4.2 Sampling Operators

The operators for graph sampling compute a subgraph by either randomly selecting a subset of vertices or a subset of edges. In addition, neighborhood information or graph traversal can be used. The computation uses a series of transformations on the input graph. Latter is stored in two datasets, one for vertices and one for edges. For each sampling operator a filter is applied to the output graph's vertex dataset to remove all zero-degree vertices following the definition of a graph sample in Section 3.

4.2.1 Random Vertex (RV) and Random Edge (RE) Sampling

Both operators RV and RE are very simple to implement. A graph with its vertex set V and edge set E serves as input for both operators. As well a sample size s has to be defined. The RV operator (Figure 1) starts with applying a filter transformation on the vertex dataset. Within the filter transformation a user-defined-function can be used to generate a random value $r \in [0, 1]$ for each vertex which is compared to the given sample size s . If the random value r is lower or equal to s , the vertex will be part of a new dataset V_1 . Otherwise the vertex will be filtered out. In the next step we join the vertices of V_1 with the edge dataset E . Within the join transformation we only select edges which corresponding source and target vertex is contained within V_1 . Those selected edges will be part of the resulting edge set E' . V' is the resulting vertex set which is equal to V_1 . The result is a graph sample containing the vertex set V' and the edge set E' .

The RE operator (Figure 2) works the other way around, as a filter transformation is applied to the edge dataset E of the input graph. An edge will be kept, again if the generated random value $r \in [0, 1]$ is lower or equal to s . After the filter transformation, all remaining edges will be stored in a new dataset E_1 . In the following we join E_1 with the input vertex dataset V . Within the join we select all source and target vertices of the edges contained within E_1 . The selected vertices are stored within the final vertex dataset V' . E' contains the final edge dataset which is equal to E_1 . The result is again a graph sample $G_S = (V', E')$.

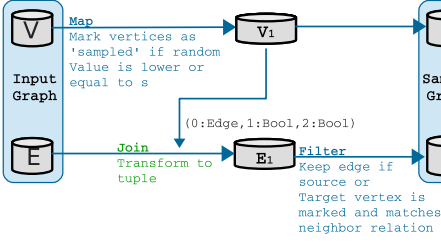


Fig. 3: Dataflow RVN Operator.

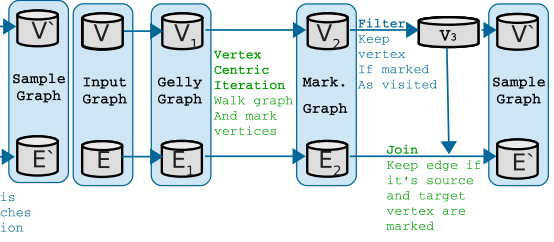


Fig. 4: Dataflow RW Operator.

4.2.2 Random Vertex Neighborhood (RVN) Sampling

As for RV and RE, a graph and a sample size s serves as input for the RVN operator (Figure 3). This approach is similar to the RV operator but also adds the direct neighbors of a vertex to the graph sample. The selection of the neighbors can be restricted according to the direction of the connecting edge (incoming, outgoing or both). In the implementation, we start with using a map transformation and randomly selecting vertices of the input vertex dataset V and mark them as sampled with a boolean flag, iff a generated random value $r \in [0, 1]$ is lower or equal than the given sample size s . In a second step, the vertex dataset V_1 is joined with the input edge dataset E , transforming each edge into a tuple containing the edge itself and the boolean flags for its source and target vertex. In an additional filter transformation on the edge tuples, we retain all connecting edges of the vertices of V_1 and apply the given neighborhood relation to create the final edge dataset E' . This relation will be either a neighbor on an incoming edge of a sampled vertex, a neighbor on an outgoing edge, or both. Note that the vertex set V' , which is equal to V_1 , of the resulting graph can contain vertices with a degree of zero. Since we choose the approach to remove all zero-degree vertices from the resulting graph sample, an additional filter transformation has to be applied.

4.2.3 Random Walk (RW) Sampling

This operator uses a random walk algorithm to walk over vertices and edges of the input graph. Each visited vertex and edges connecting those vertices will then be returned as the sampled graph. Figure 4 shows the dataflow of an input graph to a sampled graph of this operator. The input for this operator is a graph, a sample size s , an integer w and a jump probability j . At the beginning we transform the input graph to a specific Gelly format.

We are using Gelly⁵, the Google Pregel [Ma10] implementation of Apache Flink, to implement a random walk algorithm. Pregel utilizes the bulk-synchronous-parallel [Va90] paradigm to create the vertex-centric-programming model. An iteration in a vertex-centric

⁵ For more technical details see: <https://bit.ly/39UuEWS>

SF	$ V $	$ E $	Disk usage	s
1	3.3 M	17.9 M	2.8 GB	0.03
10	30.4 M	180.4 M	23.9 GB	0.003
100	282.6 M	1.77 B	236.0 GB	0.0003

Tab. 2: LDBC social network datasets.

program is called *superstep*, in which each vertex can compute a new state and is able to prepare messages for other vertices. At the end of each superstep each worker of the cluster can exchange the prepared messages during a synchronization barrier. In our operator we consider a message from one vertex to one of its neighbors a *walk*. A message to any other vertex is considered as *jump*.

At the beginning of the random walk algorithm w start vertices are randomly selected and marked as visited. The marked vertices will be referred to as *walker*. In the first superstep each walker either randomly picks one of its outgoing and not yet visited edges, walks to the neighbor and marks the edge as traversed. Or, with the probability of $j \in [0, 1]$ or if there aren't any outgoing edges left, jumps to any other randomly selected vertex in the graph. Either the neighbors or the randomly selected vertices will become the new walker and the computation starts again. Note, this algorithm is typically executed using only one walker. Since this would create a bottleneck in the distributed execution we extended the algorithm with the multi-walker approach as just explained.

For each completed superstep the already visited vertices are counted. If this number exceeds the desired number of sampled vertices, the iteration is terminated and the algorithm converges. Having the desired number of vertices marked as visited, the graph is transformed back and is now containing the marked vertex set V_2 and the marked edge set E_2 . Using a filter transformation on V_2 we create the resulting vertex set V' . A vertex will be kept if it is marked as visited. By joining V' with E_2 we only select edges which source and target vertex occur in the final vertex dataset V' and create the final edge dataset E' . The result of the operator is the graph sample $G_S = (V', E')$.

5 Evaluation

One key feature of distributed shared-nothing systems is their ability to respond to growing data sizes or problem complexity by adding additional machines. Therefore, we evaluate the scalability of our implementations with respect to increasing data volume and computing resources.

Setup. The evaluations were executed on a shared-nothing cluster with 16 workers connected via 1 GBit Ethernet. Each worker consists of an Intel Xeon E5-2430 6 x 2.5 Ghz CPU,

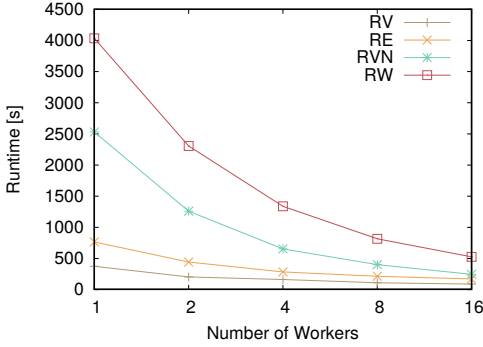


Fig. 5: Increase worker count.

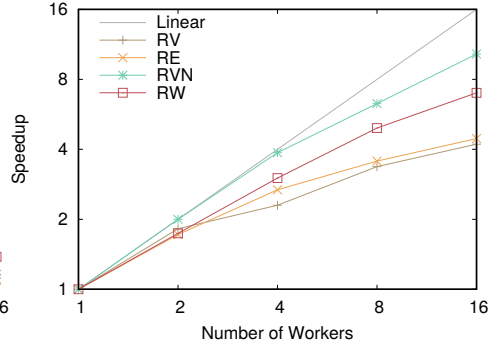


Fig. 6: Speedup over workers.

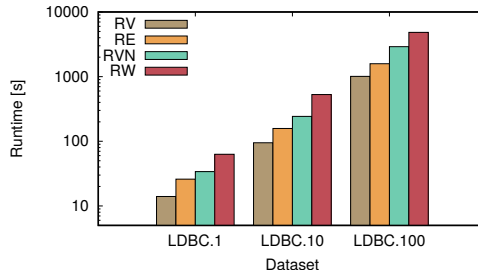


Fig. 7: Increase data volume.

48 GB RAM, two 4 TB SATA disks and runs openSUSE 13.2. We use Hadoop 2.6.0 and Flink 1.9.2. We run Flink with 6 threads and 40 GB memory per worker.

To evaluate the scalability of our implementations we use the LDBC-SNB data set generator [Er15]. It creates heterogeneous social network graphs with a fixed schema. The synthetic graphs mimic structural characteristics of real-world graphs, e.g., node degree distribution based on power-laws and skewed property value distributions. Table 2 shows the three datasets used throughout the benchmark. In addition to the scaling factor (SF) used, the cardinality of vertex and edge sets, the dataset size on hard disk as well the used sample size s are specified. Each dataset is stored in the Hadoop distributed file system (HDFS). The execution times mentioned later include loading the graph from HDFS (hash-partitioned), computing the graph sample and writing the sampled graph back to HDFS. We run three executions per setup and report the average runtimes.

5.1 Scalability

In many real-world use cases data analysts are limited in graph size for visual or analytical tasks. Therefore, we run each sampling algorithm with the intention to create a sampled

graph with round about 100k vertices. The used sample size s for each graph is contained in Table 2. As configurations, we use `Direction.BOTH` for the *RVN* algorithm and $w = 3000$ walker and a jump probability $j = 0.1$ for the *RW* algorithm.

We first evaluate the absolute runtime and scalability of our implementations. Figure 5 shows the runtimes of the four algorithms for up to 16 workers using the `LDBC.10` dataset. One can see, that all algorithms benefit from more resources. However *RVN* and *RW* gain the most. For *RVN*, the runtime is reduced from 42 minutes on a single worker to 4 minutes on 16 workers. In comparison the *RW* implementation needs 67 minutes on a single worker and 9 minutes using the whole cluster. The more simpler algorithms *RV* and *RE* are already executed relatively fast on a single machine. Their initial runtime of 405 seconds and 768 seconds on a single worker only got reduced to 105 seconds and 198 seconds.

In the second experiment we evaluate absolute runtimes of our algorithms on increasing data sizes. Figure 7 shows the runtimes of each implementation using 16 workers on different datasets. The results show that the runtimes of each algorithm increases almost linearly with growing data volume. For example, the execution of the *RVN* algorithm required about 314 seconds on `LDBC.10` and 2907 seconds on `LDBC.100`. The *RW* algorithm shows equal results with 549 seconds on `LDBC.10` and 5153 seconds on `LDBC.100`. The more simple algorithms *RV* and *RE* show very good scalability results as well.

5.2 Speedup

In our last experiment we evaluate relative speedup of our implementations on increasing cluster size. Our evaluation (Figure 6) show a good speedup for all our implementations executed on the `LDBC.10` dataset. The algorithms *RVN* and *RW* show the best speedup results of 11.2 and 8.5 for 16 workers. We assume that the usage of multiple join transformations within the *RVN* operator and the utilization of the iterative Gelly implementation used in the *RW* operator limits the speedup performance of both algorithms. However, for *RV* and *RE* we report the lowest speedup results of around 4.0. This behaviour can be explained with the already low runtimes we discovered on a single worker during the scalability evaluation in Figure 5. Hence, both operators won't benefit much of increasing cluster size, since increasing communication costs have an negative influence in both runtimes and speedup capabilities.

6 Conclusion

We outlined distributed implementations for four graph sampling approaches using Apache Flink. Our first experimental results are promising as they showed good speedup for using multiple workers and near-perfect scalability for increasing dataset sizes. In our ongoing work we will provide distributed implementations for further sampling algorithms and optimization techniques such as custom partitioning.

7 Acknowledgements

This work is partially funded by the German Federal Ministry of Education and Research under grant BMBF 01IS18026B in project ScaDS.AI Dresden/Leipzig.

Bibliography

- [Ca15] Carbone, Paris; Katsifodimos, Asterios; Ewen, Stephan; Markl, Volker; Haridi, Seif; Tzoumas, Kostas: Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
- [Er15] Erling, Orri et al.: The LDBC social network benchmark: Interactive workload. In: *Proc. SIGMOD*. 2015.
- [HL13] Hu, Pili; Lau, Wing Cheong: A Survey and Taxonomy of Graph Sampling. *CoRR*, abs/1308.5865, 2013.
- [Ju16] Junghanns, Martin; Petermann, André; Teichmann, Niklas; Gómez, Kevin; Rahm, Erhard: Analyzing Extended Property Graphs with Apache Flink. In: *Proc. ACM SIGMOD Workshop on Network Data Analytics (NDA)*. 2016.
- [Ju18] Junghanns, Martin; Kiessling, Max; Teichmann, Niklas; Gómez, Kevin; Petermann, André; Rahm, Erhard: Declarative and distributed graph analytics with GRADOOP. *PVLDB*, 11:2006–2009, 2018.
- [LF06] Leskovec, Jure; Faloutsos, Christos: Sampling from Large Graphs. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '06*, ACM, New York, NY, USA, pp. 631–636, 2006.
- [Ma10] Malewicz, Grzegorz; Austern, Matthew H.; Bik, Aart J.C.; 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. SIGMOD '10*, ACM, New York, NY, USA, pp. 135–146, 2010.
- [Va90] Valiant, Leslie G.: A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, August 1990.
- [Wa11] Wang, T.; Chen, Y.; Zhang, Z.; Xu, T.; Jin, L.; Hui, P.; Deng, B.; Li, X.: Understanding Graph Sampling Algorithms for Social Network Analysis. In: *2011 31st International Conference on Distributed Computing Systems Workshops*. pp. 123–128, June 2011.
- [Za12] Zaharia, Matei; Chowdhury, Mosharaf; Das, Tathagata; Dave, Ankur; Ma, Justin; McCauley, Murphy; Franklin, Michael J; Shenker, Scott; Stoica, Ion: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association*, 2012.
- [Zh17] Zhang, Fangyan; Zhang, Song; Chung Wong, Pak; Medal, Hugh; Bian, Linkan; Swan II, J Edward; Jankun-Kelly, TJ: A Visual Evaluation Study of Graph Sampling Techniques. *Electronic Imaging*, 2017(1):110–117, 2017.
- [ZZL18] Zhang, Fangyan; Zhang, Song; Lightsey, Christopher: Implementation and Evaluation of Distributed Graph Sampling Methods with Spark. *Electronic Imaging*, 2018(1):379–1–379–9, 2018.

Combining Programming-by-Example with Transformation Discovery from large Databases

Aslihan Özmen¹, Mahdi Esmailoghli², Ziawasch Abedjan³

Abstract: Data transformation discovery is one of the most tedious tasks in data preparation. In particular, the generation of transformation programs for semantic transformations is tricky because additional sources for look-up operations are necessary. Current systems for semantic transformation discovery face two major problems: either they follow a program synthesis approach that only scales to a small set of input tables, or they rely on extraction of transformation functions from large corpora, which requires the identification of exact transformations in those resources and is prone to noisy data. In this paper, we try to combine approaches to benefit from large corpora and the sophistication of program synthesis. To do so, we devise a retrieval and pruning strategy ensemble that extracts the most relevant tables for a given transformation task. The extracted resources can then be processed by a program synthesis engine to generate more accurate transformation results than state-of-the-art.

1 Introduction

In the era of big data, various large datasets are generated from different sources and stored in various forms. Integration and preparation of raw data from diverse sources with different schema is an important step in every data-driven analysis tasks. The preparation steps include cleaning tasks, such as normalization, entity resolution, and data transformation. In this paper, we address the vital task of data transformation discovery, which refers to the task of generating a transformation function that systematically converts values of columns from one representation to another [Ab15; Ab16; GHS12; Ji17; Mo15; Ro17; Si16].

A data transformation task might be either syntactic or semantic. Syntactic transformation tasks require syntactic manipulations on the input value via a program based on a formula or, a regular expression. A typical example is a date conversion from "XX-XX-XXXX" to "XX/XX/XXXX". Unlike the syntactic manipulation, semantic transformation cannot be performed with a formula or program and the input value only. Semantic transformations requires more context to identify an implicit relationship between the input and output values. For example, there is no formula that can calculate the airport code for a given city name. Practically one needs a lookup operation on external resources. Transformation discovery is hence a tedious task that requires domain knowledge, programming expertise, and access to external datasets. Therefore, research tried to come up with approaches to facilitate this process.

¹ TU Berlin, aslihan.ozmen@thoughtworks.com

² Leibniz Universität Hannover, esmailoghli@dbs.uni-hannover.de

³ Leibniz Universität Hannover abedjan@dbs.uni-hannover.de

There are two general directions of research for data transformation discovery. One line of research is focused on programming-by-example (PBE) approaches that learn a program to syntactically manipulate an input value and search for semantic relationships in additionally provided support tables [GHS12; Ji17; Ro17; Si16]. While PBE is highly innovative and effective on spreadsheet scale, it is bound to a small set of given and related look-up tables and cannot serve newly incoming transformation tasks. A different approach is taken by the DataXFormer system [Ab16; Mo15] and TransformDataByExample (TDE) [He18], where transformations are to be retrieved from large repositories of tables or functions. Here the given examples are used to swift through large repositories to find potential tables or functions that implement the desired hidden transformation relationship. These approaches have currently the limitation that they rely on the existence of at least one resource that fully implements the whole relationship. Consider the example depicted in Table 1. The user wants to map addresses such as “13701 Riverside Drive Pittsburgh” to the corresponding state abbreviation “PA” and at this end, the user provides a set of examples as depicted in table (a). Related tables in general purpose repositories are however unlikely to contain a resource that explicitly shows this relationship. More likely a table lists mappings of more general concepts, such as city to state as depicted in table (b). DataXFormer and TDE would both not be able to use this related table because they follow a very coarse-granular look-up approach that checks the exact match of input and output examples. Due to the large amount of candidate tables, trying to search for all possible substrings of input and output examples will hurt the performance of the transformation discovery significantly. Furthermore, it might lead to extraction of many irrelevant tables.

In this paper, we tackle exactly this problem. We want to merge the benefits of transformation discovery from large repositories and the PBE approach to harvest more accurate transformation functions. Since applying PBE on a large corpus of data is infeasible, we need to define pruning techniques that effectively limit the set of relevant tables. This problem is hard, as we do not know upfront which substrings inside the values of a table might be relevant for a transformation task at hand.

Tab. 1: Nested Syntactic transformation with lookup operation

(a) Example pair and input values		(b) Related table		
Input	Output	Town	State	County
13701 Riverside Drive Pittsburgh	PA	Pittsburgh	PA	Allegheny
27700 Medical Center Road Scarborough	?	Claremore	OK	Rogers
4270 North Blackstone Avenue Tucson	?	Scarborough	ME	Cumberland
		Tucson	AZ	Pima
		Indianapolis	IN	Marion

To this end, we propose PROTEUS that extends the DataXFormer system [Ab15; Ab16; Mo15] to be able to detect more complex transformations. PROTEUS is able to detect transformations that do not explicitly appear in the given data source. We achieve this by relaxing the exact matching approach of the DataXFormer system. We propose a multi-step filtering strategy that successively prunes irrelevant tables and cells before program synthesis

is applied. Finally, we enable parallel processing of the PBE-component to increase the scalability of our proposed system. In summary, our contributions in this paper are:

- We enhance the data transformation tool DataXFormer [Ab15; Ab16; Mo15] with the PBE framework. Therefore, our proposed system is able to detect semantically and syntactically more complex transformations.
- To make PBE feasible on millions of tables, we propose a set of pruning rules for filtering and reducing the search space to use only web tables and table entries that are relevant for the transformation task.
- By considering transformation steps per example-pair independently, we are able to process the PBE-based transformation detection task in parallel, so we are able to process the same task in the scale of millions of web tables.

2 Related Work

Several lines of research attempt to solve the task of the data transformation discovery. DataXFormer [Ab15; Ab16; Mo15] serves as the foundation of our approach. It leverages different resource types, such as Web tables [LB17; Ya12], Web forms, knowledge bases, and expert sourcing, for example-based transformation discovery. For this purpose, they proposed an inverted index for fast retrieval of relevant tables and a web form wrapper generator. The main limitation of DataXFormer is that it cannot support transformations that do not exactly match individual resource entries, i.e., tuple values of a web table.

Another line of research concerns the discovery of transformations from smaller data structures [GHS12; Ji17; SG12; Si16]. These approaches are referred to as programming-by-example (PBE) techniques, which synthesize transformation programs using input and output examples and a set of transformation operators on a small domain of data. Consequently, in case of bigger scale use cases such as web tables, they lead to real-time performance issues. In this paper, we try to combine the benefit of both lines of research. Our approach improves on DataXFormer by synthesizing transformation functions require the combination of multiple tables, by separating the retrieval process from the transformation generation process. We solve the scalability issue of PBE by defining effective pruning rules and filtering strategies.

Other than the scalability problem, systems such as REFAZER [Ro17] suffer from generalizability problem. REFAZER only uses specialized Domain Specific Language to generate codes for syntactic transformations. This system is only able to transform repetitive code transformations by learning the observations but it is not general enough to apply the transformations on pure text with unpredictable domain. Similar to Foofah, REFAZER does not apply semantic transformations at all.

Finally, there is a line of research on interactive transformation script generation [HHK15; Ji19; Ka11]. The main focus of this line of research is on interactive transformation

generation and user-support during the transformation task, which is complementary to the focus of transformation discovery for semantic transformations.

3 System Overview

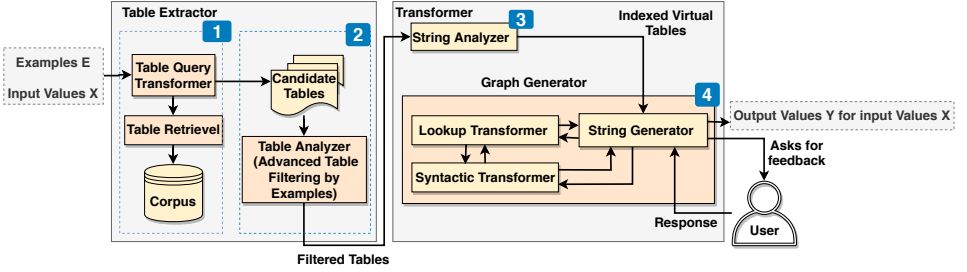


Fig. 1: System overview of PROTEUS

Figure 1 shows the overall PROTEUS architecture. PROTEUS receives a set of values $X = \{x_i | 1 \leq i \leq n\}$ that are desired to be transformed and a set of example pairs $E = \{(x_i, y_i) | x_i \in X, 1 \leq i \leq m\}$ as inputs. In the end, the system outputs the discovered transformation results Y for the remaining input values X as $\{(x_i, y_i)\}$, where y_i is the detected transformation of x_i . PROTEUS consists of two main components: *Web Table Extractor* and *Transformer*. *Web Table Extractor* is responsible for detecting the most related web tables from the table corpus and the *Transformer* component detects and generates the transformations for values in X .

In the first step, we dynamically generate queries to extract candidate tables from the corpus. The tables are indexed via an inverted index as proposed in prior work [Ab16]. The index maps tokens to tables and vice versa. Unlike DataXFormer, we do not only extract tables with exact matches but relax the extraction query to accommodate more resources and postpone the refinement to the PBE engine in *Transformer*. Because of the relaxation technique, many of the tables might be irrelevant for the transformation task at hand. Therefore, in the second step, we filter candidate tables that are unlikely to support the transformation task. In the third step, we send the relevant candidate tables to the *Transformer* component. *Transformer* first applies finer-granular pruning rules and scoring functions to remove potential noisy, irrelevant, and duplicate records from the extracted tables. For this purpose it uses a row-level filtering approach to detect the most relevant entries inside each selected external table. In the fourth step, we leverage the PBE framework to generate syntactic manipulation programs for the relevant tables. The programs can be represented as graphs of operations that connect input values to the output values of the provided examples in E . The common path between all these graphs is the answer for the transformation task.

The graph generation for each example pair can be performed in parallel, which allows us to scale the process for more extracted tables. If there is not a common path, the user can

choose the right graph among two disjoint graphs. In the end, PROTEUS applies the final chosen graph as the desired program to generate the output for the remaining input values.

4 Web Table Extractor

The *Web Table Extractor* takes example pairs (E) and returns transformation-related web tables. It consists of the two sub-components: *Table Query Transformer* and *Table Analyzer*.

The **Table Query Transformer** is implemented as an extension of the query generator of DataXFormer. The original DataXFormer query generator creates a single SQL query with two IN operators, one for the input values and one for the output values to identify tables that contain at least τ example pairs. To obtain more data, we relax this query twofold. First, PROTEUS tokenizes every input and output value from the example pairs and uses each generated token independently for table retrieval. For instance, if the given example-pair is “13701 Riverside Drive Pittsburgh \rightarrow PA”, it generates queries to find tables that contain at least one token from the input (here: “13701”, “Riverside”, “Drive”, “Pittsburgh”) and one token from the output (here: “PA”). If no table is found, the system generates a new query to find tables that contain any of the tokens as a substring of at least one entry. This is done leveraging the LIKE predicate in SQL. The first query often returns a large number of web tables, in which case we drop the time consuming query with the LIKE predicate.

A straightforward approach to detect the desired transformation would be to evaluate every single retrieved table from the *Table Query Transformer* sub-component. This approach is time-consuming and error-prone because of the large number of irrelevant tables. Therefore, **Table Analyzer** further reduces the search space by filtering the irrelevant tables. First, it determines whether at least the row alignment for input/output pairs in each table is correct [Ab16]. Input and output pairs in each example should appear in the same rows. While DataXFormer checks the row alignment for exact input/output values, we check the row alignment of partial tokens and substrings of the input/output values. Note that this alignment is necessary for the program generation later in order to obtain consistent program paths that connect the input value to the output value. Furthermore, we also want to get rid of tables where the tokens are randomly aligned. Especially in long column values the occurrence of multiple tokens is likely. Thus we also drop tables where the Jaccard similarity of the aligned rows and the corresponding original input examples is below 50%.

The remaining selected tables that pass the alignment test, will be rated based on their relatedness to the transformation task. Depending on the number of matches and the strength of the matches, tables differ in the degree of relatedness to the transformation task. Similar to DataXFormer, PROTEUS leverages a *refine* technique based on expectation - maximization (EM) to update the scores of tables and found alignments. However, our approach differs in two ways. First, we also accommodate the fact that not every token is equally important and further, we are only interested in obtaining the scores while DataXFormer uses the final scores to choose the transformation. In the expectation step the scores of the tables are

updated and in the maximization step the scores of the instances. In each step the scores of the other step is used to update the scores. This process converges as soon as the the updates are below a very low threshold ϵ . The process starts with scores that reflect the ratio of existing example pairs inside the extracted tables. However, as we are considering partial values, i.e., tokens, we also consider the fact that not every token is equally important. For instance, if there is a match for value “of” in table T_1 and another match for value “New York City” in table T_2 . It is desirable to give more weight to the table that contains “New York City” because it is more specific and thus more likely to support our specific task. To reflect this property in the EM formula, we use the Inverse Document Frequency (IDF). Higher IDF score means that the token is more specific. Therefore in the EM model, we multiply the score of each occurring example with its IDF. Finally, *Table Analyzer* sends the tables along their calculated scores to the *Transformer* component.

5 Transformer

The *Transformer* component has two sub-components: *String Analyzer* and *Graph Generator*.

String Analyzer takes the tables from the *Web Table Extractor* sorted by their relatedness scores and finds the most relevant table entries (rows) for each input value. Each input value can be matched with more than one entry inside a table. Therefore, *String Analyzer* reduces them to the most promising match to unburden the PBE step from generating programs for the matches that are less likely to be a candidate for the transformation.

To find the most related entry to the input value of an example pair, we leverage a score based on the Longest Common Substring (LCS) [AO11]. This is consistent with most PBE algorithms, which use common string patterns between two strings to detect the transformations. We calculate the LCS score between each input value from the example pairs and all the matched entries in the related tables. The table entry with the maximum LCS score is selected to be used in the program generation phase. If there are two table entries with the same LCS score, the edit distance, a.k.a. Levenshtein distance [Le66] will break the tie. The entry with lower edit distance will be picked as more related.

Consider the example in Table 2. The first two example pairs in the left table are provided by the user who wants to transform the address of the last two inputs to their corresponding state codes. The table on the right reflects a candidate web table, which contains aligned input/output tokens of our examples (the result of the steps before). In this candidate table, the first row and the value “Redmond” has the highest LCS with the given input example “1906 Jackson Way Redmond”. The same way, we compute the LCS for the second example pair. In this particular case, there are two entries with the same LCS, the third and the fifth rows contain both entries with the LCS length of 8. Here, the similarity check would be the tie breaker, which chooses the entry in the third row. The String Analyzer outputs the top related tables and the injective mapping of example pairs to rows in each table.

Tab. 2: Simple LCS and similarity check example in *Transformer*

(a) Input values (X) including the example pairs (E)		(b) Related web table	
Input	Output	C1	C2
1906 Jackson Way Redmond	WA	Redmond	WA, Washington
1703 Red Creek Road Richmond	VA	Redm. branch is open till 24/02	WA , Washington
101 Sundown Blvd Sacramento	?	Richmond	VA, Virginia
510 Green Lake Road Beaumont	?	Beaumont	TX, Texas
		Richmond branch is open only till 03/03	VA, Virginia
		Sacramento	CA, California

Graph Generator discovers the programs that convert an input value to a provided output transformation following the PBE approach [GHS12]. This sub-component receives the final set of tables, top related entries, and the corresponding example pairs and then generates the corresponding Directed Acyclic Graph (DAG) for each example pair. In this graph each node represents a transformation state of a value and each edge corresponds to a program that changed the value of its source node to the value in the target node. The PBE approach combines programs for syntactic manipulations, which are concatenations of regular expressions based on sub-strings and table mappings. For instance, a syntactic manipulation would be a program that converts “Bob.Franklin@tu-berlin.de” to “Bob Franklin” by learning to remove the substring after “@” and replace the first dot symbol by a space. Depending on the provided resources, several possible paths can be generated to map an input example to its output value.

Considering our running example, the graph for the first example pair shown in Figure 2 is generated using the first row in the web table, because based on the LCS score, the first row in the web table is the closest to the example pair. Edges with the labels of $Prog_i$ represent the generated code snippets and η_j represents the value in the j^{th} node. Here, $Prog_1$ represents the program to extract the first alphabetic string after the last space from the input value. The next program encodes the mapping from “Redmond” to “WA, Washington”. And finally, $Prog_3$ extracts the first alpha-numerical substring from “WA, Washington”.

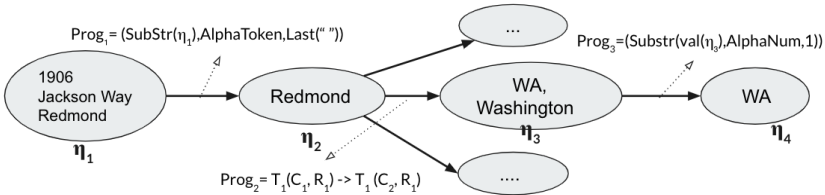


Fig. 2: Generated graph for the first example pair in *Transformer*

Graph Generator greedily starts with the examples inside the table with the highest relatedness score and starts to generate program graphs for them. Whenever it generates the

graphs of the next example pair it starts to find the intersection of the two sets of graphs and this way gradually identifies the graph that covers all example pairs.

In the end, the intersection of all paths will be the program that can be used to transform the remaining inputs. Once all the input values are transformed, all tables have been processed, or the result of graph intersection is empty, the algorithm stops. The latter suggests that no transformation could be found that covers all example pairs but there are still other tables that might contain a fitting graph path. In such cases, PROTEUS can return possible graph candidates and ask the user to choose the correct one. PROTEUS can then proceed with the chosen graph to find a match for remaining input values in the remaining tables. Figure 3 shows the application of graph intersection on our running example.

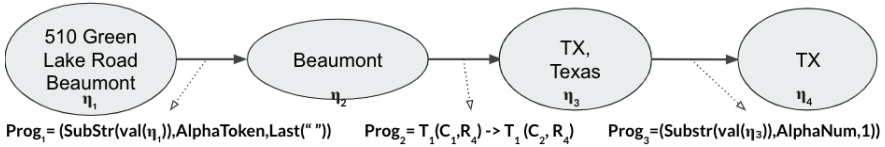


Fig. 3: Path after the result of the final intersection

We generate the graphs independently from each other so that we are able to run each graph generation and graph intersection in parallel. For instance, while we simultaneously generate graphs for first and second example pairs, once we start intersecting them, we generate the third example pair at the same time and so on. Defining the transformation tasks as sub-independent processes enables us to perform transformations in parallel.

6 Evaluation

In this section, we show the efficiency and effectiveness of our transformation discovery system and compare it to the state-of-the-art.

6.1 Data and experimental setup

We use the Dresden Web Table Corpus⁴ [Eb15] as the table corpus to extract transformation-related tables. It contains 145 million web tables from about 4 billion web pages. We evaluate our approach by running 20 semantic data transformation tasks. Each transformation task has 3 unique example pairs. Four of these transformation tasks (first two rows in Table 3) are created manually to ensure that these tasks require syntactic manipulations before and after lookup operations e.g., “*Country is Ukraine* → *City name: Kiev*”. The rest of the tasks are public benchmarks found in Bing Search Engine query logs and asked by Data Scientists and BI Analysts in StackOverflow. Table 3 shows all the transformation tasks used in this paper. We compare our system to four state-of-the-art approaches: DataXFormer [Ab15;

⁴<https://wwwdb.inf.tu-dresden.de/misc/dwtc/>

Tab. 3: Transformation tasks used in this paper

Input	Output	Input	Output
Country Names	Capitals	Country Names with Attributes	Capitals
Element names	Boiling point	Country names	Denonyms
Color Number	Color code	Regular time	Military Time
Hijri	Gregorian Calendar	Company Address	State
MB	GB	yyyymmdd	Datetime
Number	Numeric Padding	String	Camelize Casing
Regular Format	ISBN Format	Datetime	Month
Cookies	Domain name	Month number	Month name
CUSIP	Ticker	Product	Company
MEME	Filename	Time span	hrs mins secs

Ab16; Mo15], Gulwani’s approach [GHS12], FlashFill [Gu16], and Foofah [Ji17]. We ran our experiments on a machine with 2.9 GHz Intel Dual Core CPU and 8 GB RAM. Codes are available in our GitHub repository⁵.

6.2 Results

Coverage. We define coverage as the ratio of the number of transformation tasks where the system returns a correct transformation output for at least one of the input values. As depicted

in Table 4, PROTEUS achieves considerably higher coverage than the other baselines. It achieves 95% coverage, which means that our system generates output for 19 out of the 20 transformation tasks. High coverage conveys the fact that PROTEUS is more robust to the noisy data which is common in web tables. Noisy and erroneous data results in lower exact match rate and in the end, it will lead to less related tables and lower coverage rate. Foofah, due to the lack of ability in Regex matching, DataXFormer because of only using exact match, Flashfill because of the lack of lookup operation and being limited to only program generation, and Gulwani’s system due to the fact that it requires clean and user-defined tables for the transformation tasks, have low coverage. The only task that was not covered by PROTEUS was “hijri to the gregorian calendar”. This conversion is difficult based on static web tables. Consider the following input/output example pair given by the user:

“11 Shawwal 1430” → “Wednesday 30 September 2009 C.E”

PROTEUS cannot find web tables that contain the values “11 → 30” and “1430 → 2009”.

Effectiveness. As shown in Table 4, PROTEUS outperforms other systems in terms of Precision and recall. Leveraging token-based matching that retrieves wider range of candidate tables from the corpus improves the recall. More candidate tables increase the

Tab. 4: Coverage, Precision, and Recall for 20 tasks.

Systems	#Coverage	Precision	Recall
PROTEUS	95% (19/20)	90%	78%
FlashFill	50% (10/20)	50%	50%
Gulwani	50% (10/20)	50%	50%
DataXFormer	30% (6/20)	24%	17%
Foofah	5% (1/20)	5%	5%

⁵<https://github.com/aslihanozmen/Proteus>

chance to find the desired transformation. On the other hand, the LCS-based entry filtering strategy allows the system to pick the most fitting table entries to the input values and drop the irrelevant ones. Therefore, the final transformations are only generated using the most promising candidates improving the precision. Gulwani’s approach because of being limited to the provided tables, and FlashFill and Foofah because of their limitation to syntactic transformations achieve lower precision and recall.

Runtime. As shown in Figure 4, PROTEUS is faster and more scalable than DataXFormer and Gulwani’s approach. We excluded FlashFill and Foofah from the runtime evaluation because they only cover syntactic manipulations based on the input values. PROTEUS is fast due to its pruning rules, and parallelization of the graph processing. It is also faster than DataXFormer because before any look up operations, PROTEUS checks whether syntactic manipulations of the input values alone can perform the transformation for all the input values. In these five cases it refrained from looking for more complicated semantic transformations. These five tasks are “MB \rightarrow GB”, “Cookies \rightarrow Domain name”, “Regular format \rightarrow ISBN format”, “Number \rightarrow Numeric padding”, and “String \rightarrow Camelize casing”.

String Analyzer. Our LCS-based entry scoring technique that only keeps the top related entry, reduces the number of generated *Progs* more than 50 times compared to the raw PBE approach. This reduction is due to the elimination of the irrelevant table entries before the program generation phase. As an example, for the transformation task “CUSIP” \rightarrow “Ticker”, without using our LCS-based pruning, the first two generated graphs for the first two examples contain overall 44,075 *Progs* (edges). Intersecting these two graphs leads to a graph with 9,276 *Progs*. Applying the LCS pruning, the total number of *Progs* for the first two generated graphs decreases to 750 and their intersection to 8 *Progs*.

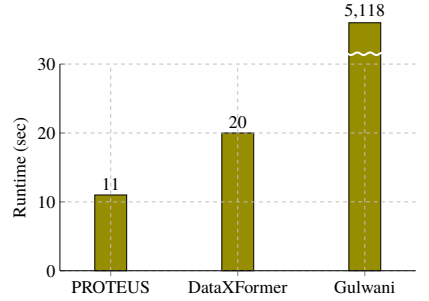


Fig. 4: Runtime comparison

7 Conclusion

In this paper, we proposed a system to combine transformation discovery with programming by example. The general idea was to first relax the transformation discovery systems to obtain more partially relevant resources and then filter irrelevant resources with PBE. Our experiments show that the approach is covering more transformation tasks than state-of-the-art. Unlike the state-of-the-art, PROTEUS has higher robustness to noisy data that is very common in web tables. In future, we would like to make the filtering steps more dynamic to avoid heuristic-based thresholds.

Acknowledgements. This project has been supported by the German Research Foundation (DFG) under grant agreement 387872445.

References

- [Ab15] Abedjan, Z.; Morcos, J.; Gubanov, M. N.; Ilyas, I. F.; Stonebraker, M.; Papotti, P.; Ouzzani, M.: Dataxformer: Leveraging the Web for Semantic Transformations. In: CIDR. 2015.
- [Ab16] Abedjan, Z.; Morcos, J.; Ilyas, I. F.; Ouzzani, M.; Papotti, P.; Stonebraker, M.: DataXFormer: A robust transformation discovery system. In: ICDE. Pp. 1134–1145, 2016, URL: <https://doi.org/10.1109/ICDE.2016.7498319>.
- [AO11] Arnold, M.; Ohlebusch, E.: Linear Time Algorithms for Generalizations of the Longest Common Substring Problem. *Algorithmica* 60/4, pp. 806–818, 2011, URL: <https://doi.org/10.1007/s00453-009-9369-1>.
- [Eb15] Eberius, J.; Braunschweig, K.; Hentsch, M.; Thiele, M.; Ahmadov, A.; Lehner, W.: Building the Dresden Web Table Corpus: A Classification Approach. In: BDC, 2015. IEEE Computer Society, pp. 41–50, 2015, URL: <https://doi.org/10.1109/BDC.2015.30>.
- [GHS12] Gulwani, S.; Harris, W. R.; Singh, R.: Spreadsheet data manipulation using examples. *Commun. ACM* 55/8, pp. 97–105, 2012, URL: <https://doi.org/10.1145/2240236.2240260>.
- [Gu16] Gulwani, S.: Programming by Examples: Applications, Algorithms, and Ambiguity Resolution. In: *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*. Springer-Verlag, Berlin, Heidelberg, pp. 9–14, 2016, ISBN: 978-3-319-40228-4, URL: https://doi.org/10.1007/978-3-319-40229-1_2, visited on: 03/10/2019.
- [He18] He, Y.; Chu, X.; Ganjam, K.; Zheng, Y.; Narasayya, V. R.; Chaudhuri, S.: Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11/10, pp. 1165–1177, 2018, URL: <http://www.vldb.org/pvldb/vol11/p1165-he.pdf>.
- [HHK15] Heer, J.; Hellerstein, J. M.; Kandel, S.: Predictive Interaction for Data Transformation. In: CIDR. 2015, URL: http://cidrdb.org/cidr2015/Papers/CIDR15%5C_Paper27.pdf.
- [Ji17] Jin, Z.; Anderson, M. R.; Cafarella, M. J.; Jagadish, H. V.: Foofah: Transforming Data By Example. In: SIGMOD. ACM, pp. 683–698, 2017, URL: <https://doi.org/10.1145/3035918.3064034>.
- [Ji19] Jin, Z.; Cafarella, M. J.; Jagadish, H. V.; Kandel, S.; Minar, M.; Hellerstein, J. M.: CLX: Towards verifiable PBE data transformation. In: EDBT. Pp. 265–276, 2019.
- [Ka11] Kandel, S.; Paepcke, A.; Hellerstein, J. M.; Heer, J.: Wrangler: interactive visual specification of data transformation scripts. In: CHI. Pp. 3363–3372, 2011.
- [LB17] Lehmberg, O.; Bizer, C.: Stitching Web Tables for Improving Matching Quality. *PVLDB* 10/11, pp. 1502–1513, 2017.

- [Le66] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. Vol. 10. 8, pp. 707–710, 1966.
- [Mo15] Morcos, J.; Abedjan, Z.; Ilyas, I. F.; Ouzzani, M.; Papotti, P.; Stonebraker, M.: DataXFormer: An Interactive Data Transformation Tool. In: SIGMOD. ACM, pp. 883–888, 2015, URL: <https://doi.org/10.1145/2723372.2735366>.
- [Ro17] Rolim, R.; Soares, G.; D’Antoni, L.; Polozov, O.; Gulwani, S.; Gheyi, R.; Suzuki, R.; Hartmann, B.: Learning syntactic program transformations from examples. In: Proceedings of ICSE. IEEE, pp. 404–415, 2017, URL: <https://doi.org/10.1109/ICSE.2017.44>.
- [SG12] Singh, R.; Gulwani, S.: Learning Semantic String Transformations from Examples. CoRR abs/1204.6079/, 2012, arXiv: 1204.6079, URL: <http://arxiv.org/abs/1204.6079>.
- [Si16] Singh, R.: BlinkFill: Semi-supervised Programming By Example for Syntactic String Transformations. Proc. VLDB Endow. 9/10, pp. 816–827, 2016, URL: <http://www.vldb.org/pvldb/vol9/p816-singh.pdf>.
- [Ya12] Yakout, M.; Ganjam, K.; Chakrabarti, K.; Chaudhuri, S.: Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In: SIGMOD. Pp. 97–108, 2012.

Towards Learned Metadata Extraction for Data Lakes

Sven Langenecker¹ Christoph Sturm² Christian Schalles³ Carsten Binnig⁴

Abstract: An important task for enabling the efficient exploration of available data in a data lake is to annotate semantic type information to the available data sources. In order to reduce the manual overhead of annotation, learned approaches for automatic metadata extraction on structured data sources have been proposed recently. While initial results of these learned approaches seem promising, it is still not clear how well these approaches can generalize to new unseen data in real-world data lakes. In this paper, we aim to tackle this question and as a first contribution show the result of a study when applying Sato — a recent approach based on deep learning — to a real-world data set. In our study we show that Sato without re-training is only able to extract semantic data types for about 10% of the columns of the real-world data set. These results show the general limitation of deep learning approaches which often provide near-perfect performance on available training and testing data but fail in real settings since training data and real data often strongly vary. Hence, as a second contribution we propose a new direction of using weak supervision and present results of an initial prototype we built to generate labeled training data with low manual efforts to improve the performance of learned semantic type extraction approaches on new unseen data sets.

Keywords: data lakes; dataset discovery and search; semantic type detection

1 Introduction

Motivation: Data lakes are today widely being used to manage the vast amounts of heterogeneous data sources in enterprises. Different from classical data warehouses, the idea of data lakes is that data does not need to be organized and cleaned upfront when data is loaded into the warehouse [Di14]. Instead, data lakes follow a more “lazy” approach that allows enterprises to store any available data in its raw form. This raw data is organized and cleaned once it is needed for a down stream task such as data mining or building machine learning models. However, due to the sheer size of data in data lakes and the absence (or incompleteness) of a comprehensive schema, data discovery in a data lake has become an important problem [Ma17; Na20; RZ19].

One way to address the data discovery problem, is to build data catalogs that allow users to browse the available data sources [Na19]. However, building such a data catalog manually would again pose high effort since metadata needs to be annotated on data sources. An important task for cataloging structured (table-like) data in a data lake (e. g., originating from CSV files) is to derive semantic type information for the different columns of a data

¹ DHBW Mosbach, Germany sven.langenecker@mosbach.dhbw.de

² DHBW Mosbach, Germany christoph.sturm@mosbach.dhbw.de

³ DHBW Mosbach, Germany christian.schalles@mosbach.dhbw.de

⁴ TU Darmstadt, Germany carsten.binnig@tu-darmstadt.de

set. The reason is that this information is often missing in many data sources or the column labels available in data sources are not really helpful for data discovery since they use technical names or have been annotated from users with a different background.

In order to tackle the problem of extracting semantic data types from structured data sources in data lakes, recently learned approaches for metadata extraction have been proposed [Ca18b; Hu19; Zh20]. The main idea of these learned approaches is that they use a deep learning model for semantic type detection where the models are trained on massive table corpora with already annotated columns. While initial results of these learned approaches seem promising, it is still not clear how well these approaches can deal with the variety of data in real data lakes.

Contributions: In this paper, we aim to tackle this question and report on our initial results of analyzing the quality of the state-of-the-art learned approaches for metadata extraction on real-world data. Moreover, we also show initial results of a new direction of tackling the open problems of the learned approaches that we discovered in our analyses. In the following, we discuss the two main contributions of this paper.

As a first contribution, we show the result of a study when applying Sato [Zh20] - a recent approach based on deep learning to extract semantic types - to a real-world data set. A inherent problem of deep learning-based approaches for semantic type extraction is that they rely on a representative training data set; i.e., a set of columns with labeled semantic types. Otherwise, if the training data set does not cover the broad spectrum of data characteristics and types, the performance of the learned models quickly degrades when applied to a new data set. In fact, we show that Sato without re-training was only able to extract semantic data types for about 10% of the columns on the data sets used in our study.

As a second contribution, we thus suggest to take a new direction for learned metadata extraction to tackle the shortcomings of the existing deep learning approaches. As mentioned before, the root cause of why existing approaches often fail to extract semantic types is that the training data of the learned approaches is too narrow and thus the performance on new data sets is often poor. Hence, in this paper we propose a new direction of using weak supervision to generate a much broader set of labeled training data for semantic type detection on the new data set. Our initial results show that our approach can significantly boost the performance of deep learning-based approaches such as Sato when re-training these approaches on the additional synthesized training data.

Outline: In Section 2, we first provide an overview of approaches for metadata extraction from structured data in data lakes. Afterwards, we discuss the results of our study of using Sato as a recent learned approach on a real-world data set in Section 3. Moreover, we then discuss our new approach based on weak supervision in Section 4. Finally, we present the initial results of using our current prototype in Section 5 before we conclude in Section 6.

2 Overview of Existing Approaches

In the following, we give a short overview of selected existing approaches for metadata extraction. We first discuss approaches for semantic type extraction before we briefly summarize recent approaches for the extraction of relationships.

2.1 Extraction of Semantic Types

Approaches that automatically extract types from metadata of data sources are already well established in industry. Prominent examples are Azure Data Catalog [Az20], AWS Glue [AW20] and GOODS [Ha16]. In addition, many other research efforts exist for developing generic metadata models and special algorithms for metadata extraction (e. g. [QHV16]).

All these approaches rely on the fact that basic metadata information is already annotated in the data source (e. g., as a header row in a CSV file) such as column and table names. However, header rows exist only in few cases and even when they do, the attribute names are not always useful as a semantic type. In this case, existing systems opt for manual metadata annotation.

Considering the huge amount of heterogeneous, independent, quickly changing data sources of real-world data lakes these approaches reach their limit. Therefore, some systems aim to detect semantic types from the columns content instead of relying on already existing labels in the sources. For this purpose, there exist two main research directions for automatic semantic type detection: search-based approaches and learning-based approaches.

Search-based Approaches: The main idea of search-based approaches is to use external information to annotate semantic information to data sets. One approach in this direction is AUTOTYPE [YH18] which searches for existing custom extraction code to handle more specific (domain dependent) semantic data types. By helping developers to find and extract existing type detection code the supported semantic types of AUTOTYPE can be extended semi-automatically.

Learning-based Approaches: In contrast to search-based approaches, learning-based approaches aim to build a machine learning model that can derive semantic types of columns from example data and not from extraction code. An early approach in this class is [LSC10] which uses machine learning techniques to annotate web tables and their columns with types. While this approach relies on graphical models for extracting semantic labels for columns, more recent approaches such as [Hu19; Zh20] are based on a deep neural network.

Sato [Zh20] which is the successor of Sherlock [Hu19] thus requires training data with labeled semantic types. While Sherlock only uses the individual column values as features for predicting the semantic type, Sato also uses context signals from other columns in the table to predict the semantic type of a given column.

2.2 Extraction of Relationships

While extraction of semantic types is one important direction for metadata extraction, there also exist other approaches that are able to derive relationships between datasets (e. g., an author *writes* a book) from data automatically. One prominent example for such an approach is AURUM [Ca18a]. While AURUM represents an overall system for building, maintaining and querying an enterprise knowledge graph for available data sources, SEMPROP [Ca18b] is the subsystem of AURUM, which automatically derives links (i.e., relationships) between the data sources using word embeddings.

As mentioned before, different from this work and similar to Sato in this paper we focus on the extraction of semantic types for structured data sets. Hence, in the following sections we will limit the analysis of learned approaches to this direction. However, extending our approach towards relationship extraction is an interesting avenue of future work.

3 Study of Using Learned Approaches

In the following, we present the results of our study of using learned semantic type extraction approaches on real-world data. For our study, we selected Sato [Zh20] as a recent approach based on deep learning.

3.1 Data Sets and Methodology

Data Sets: As a data set in this study, we use the Public BI Benchmark⁵ data corpus. The data corpus contains real-world data, extracted from the 46 biggest public workbooks in Tableau Public⁶. In this corpus there are 206 tables each with 13 to 401 columns. The main reason for choosing this corpus for our study was that it contains labeled structured data from different real-world sources across various domains (e. g. geographic, baseball, health, railway, taxes, social media, real estate). Hence, the benchmark comes with a high diversity and heterogeneity of data sources that can typically also be found in data lakes of enterprises today.

Methodology: As mentioned before, the inherent problem of deep learning-based approaches for semantic type extraction is that they rely on a representative training data set. To put it differently, if the training data set does not cover the variety of cases that are also seen in the real-world data, the performance of the learned models quickly degrades. As part of our analysis, we wanted to see to which extent this inherent limitation influences the overall quality of a learned approach such as Sato.

For the study, we thus annotated the data in the Public BI Benchmark manually with the correct semantic types of Sato. For the annotation, we first preprocessed the data automatically and searched for string matches between the column headers of the tables in the Public BI Benchmark and the semantic types supported by Sato. To guarantee the

⁵ https://github.com/bogdanghita/public_bi_benchmark-master_project

⁶ <https://public.tableau.com>

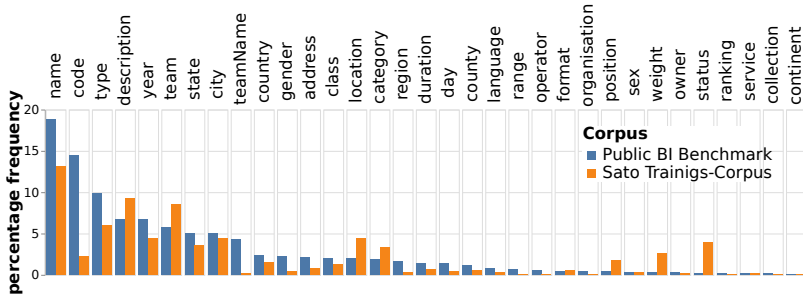


Fig. 1: Distribution of semantic types in training data of Sato and the Public BI Benchmark

correctness of labels every column was additionally inspected and missing types were added manually.

3.2 Results of the Study

As a first question, we analyzed the coverage rate of the 78 semantic types supported by Sato in the Public BI Benchmark to see to which extend a pre-trained model can support real-world data if no new training data is used for re-training. For this question, we analyzed what fraction of columns in the Public BI Benchmark had a type that was covered by the training data set of Sato. The main result of this analysis was that only 10.6% of the columns are assignable to one of the semantic types.

As a second question, for the columns of the Public BI Benchmark that have types which are supported by Sato, we then wanted to see how the distribution of the 78 semantic types in the training data used for Sato and the Public BI Benchmark look like. The reason is that different distribution of labels in the training and testing data can have a negative impact on the overall quality of a learned approach. As can be seen in Fig. 1, the frequency for many semantic types in both data sets (i.e., original training data of Sato and the Public BI Benchmark), however, is almost identical.

As a final question, we thus aimed to analyze the 10.6% of the columns in the Public BI Benchmark that are in principle covered by the training data of Sato. For this, we used the pre-trained Sato model and applied it to only this fraction of the data of the Public BI Benchmark. For this subset, Sato achieves an F_1 score (macro average and weighted⁷) of 0.090 and 0.300 respectively, which is also shown in Tab. 1 in our evaluation in Section 5. The original paper [Zh20] reports an F_1 score of 0.735 and 0.925 on the VizNet⁸ data corpus. This indicates that the data characteristics of the supported data types of the Public BI Benchmark is different from the data characteristics of the training data of VizNet and thus Sato can not infer types in a robust manner (even if they should be supported in principle).

⁷ **F_1 score macro average:** averaging the unweighted mean F_1 score per label

F_1 score weighted average: averaging the support-weighted mean F_1 score per label

⁸ <https://github.com/mitmedialab/viznet>

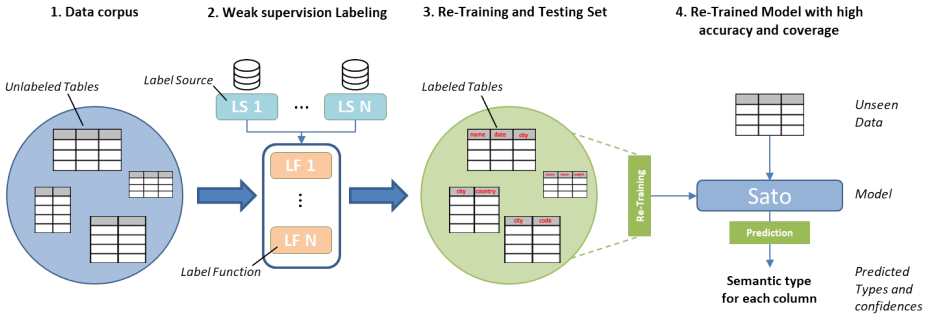


Fig. 2: Concept and step-by-step procedure of our weak supervision approach

Main Insights: As suspected, our study has shown that a deep model such as Sato trained on one data set can only cover a fraction of data types of a new data set. Moreover, for the overlapping data types, the accuracy is still pretty low due to different data characteristics of the training data and the new data set. While the results of our study are specific to Sato, we believe that our findings are much more general and translatable to any learned approach that relies on manually curated training data (which is inherently limited as discussed before). Hence, a new approach is required where one can easily adapt learning-based models for type extractors to new data sets that covers types and data characteristics not covered in the available manually labeled training data. As a solution for this requirement, we next present our new weak supervision approach in the next section.

4 Weak Supervision for Semantic Type Extraction

The root cause of why deep learning-based approaches such as Sato often fail to extract semantic types on a new data set is that the training data lacks generality as discussed before. The main idea of using weak supervision is to generate a broad set of labeled training data with only minimal manual effort and thus increase the robustness when applying a learned approach such as Sato to a new data set. In the following, we discuss our initial ideas for such an approach and present the first results of our prototype to showcase its potential.

4.1 Overview of Our Approach

Fig. 2 shows an overview of our approach. The main idea is that based on a set of simple labeling functions, we generate new (potentially noisy) training data that is then used to re-train a model such as Sato to increase the coverage of data types and data characteristics of the learned model. In other words, we apply the ideas of data programming discussed in [Ra17] for the domain of semantic type extraction.

For generating new training data in our approach, we differentiate between two different classes of labeling functions: (1) The first class are labeling functions that can generate labels (i.e., semantic types) for completely new semantic types in a data lake that are not yet covered by a manually labeled training data set. Labeling functions of this class can

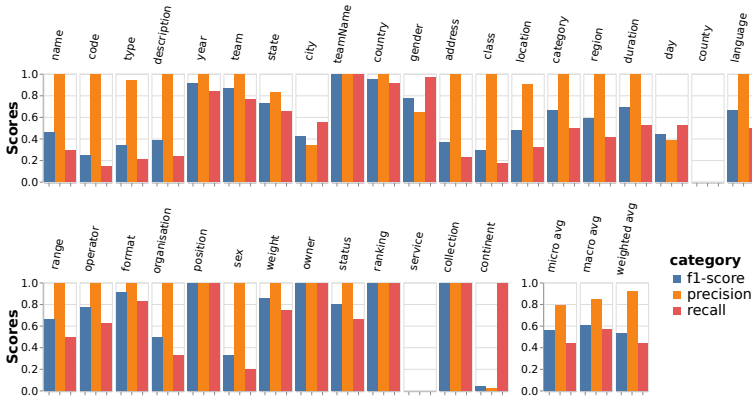


Fig. 3: Performance of clustering semantically similar columns

be, for example, regular expressions, dictionary lookups, or other techniques such as using alignment with existing ontologies. (2) Second, as we have seen in our study, another problem of learned approaches such as Sato is that they often fail to predict semantic types even if in principle the semantic type is already covered by the training data. The main reason for this case is that the training data does not cover the wide spectrum of data characteristics that might appear in a new data set. Hence, as a second class of labeling functions we support functions that can generate new labeled columns that cover more data characteristics (e.g., new values) for data types that are already available in a training data set. One idea for a labeling function of this class is the use of word embeddings [Mi13] to cluster new unlabeled with already labeled columns and thus generate new labeled columns for existing semantic types. A more detailed description of such a labeling function is given below.

4.2 Label Generation using Clustering

For generating more labeled training data for an existing semantic type, we implemented a method based on clustering in our prototype system that we briefly introduced before. As mentioned, the main idea is that we can start with a small training corpus of labeled columns and by clustering new non-labeled to the labeled columns, we can derive new labeled training data.

To implement this labeling approach, we first compute column embeddings for labeled and unlabeled columns based on word embeddings of individual values. As word embeddings, we currently use *Google USE*⁹ that was trained on 16 different languages and showed good results. But in principle we could also use other word embeddings. Based on the embeddings of individual values, we compute an embedding for all values of a column by calculating the average across the embeddings of all values which is the dominant approach for building representations of multi-words also mentioned in other papers [So12]. This approach is

⁹ <https://tfhub.dev/google/universal-sentence-encoder-multilingual-large/3>

reasonable also for us, since string-typed column values in the Public BI Benchmark are only composed of single values. In general, in the case the column values themselves consist of a sequence of words, we could also consider word embedding combining techniques as represented in [LM14] or [Ca18b].

Once we computed an embedding for all values of a column, we next carry out the clustering of labeled and unlabeled columns based on these embeddings. For this step, we use an agglomerative clustering algorithm¹⁰. In our prototype, we use this clustering method to not generate a fixed number of clusters, but to form groups based on the cosine similarity of vectors (i.e., our embeddings) and a distance threshold that we discuss below. Once clustered, we then compute a semantic type per cluster based on the majority vote of columns with the same label. [Ma19] represents a system called Raha, which relies on a similar idea for generating training data but for error detection and not for semantic type extraction.

A key parameter to be set in our clustering approach is the distance threshold which can vary between 0.0 and 1.0 (i.e., a lower value means that we produce more clusters). In our experiments, we used a threshold of 0.1 based on a hyper-parameter search on the already labeled columns. This threshold provided high accuracy on the broad spectrum of data sets in the Public BI Benchmark.

Initial Results: To analyze if the basic idea of clustering is working, we conducted a small experiment where we measure how well the clustering approach works on the Public BI Benchmark using our annotations of the 78 Sato types. By clustering, we wanted to see whether columns with the same type would be assigned to the same cluster. As we see in Fig. 3, with a few exceptions, the clustering algorithm achieves high precision. This means that there is a very high probability that all elements in one cluster belong to the semantic type representing the specific cluster. For many types, we achieve an F_1 score of 1.0 such as for the semantic types *teamName*, *position*, *owner*, *ranking* and *collection*.

Moreover, in a second experiment, we wanted to show the robustness of our clustering approach to different data characteristics. For showing this, we analyzed the entropy and the jaccard-coefficient for all columns with the same semantic type in the Public BI Benchmark. The intuition is that columns with a high entropy (i.e., a high degree of divergence) or pairs of columns which have a low jaccard-coefficient (i.e., where columns values are not overlapping) are harder to cluster. Overall, our approach assigns column pairs with the same semantic type to the very same cluster even if they strongly vary in the entropy or have a low overlap (i.e., a low jaccard-coefficient). Unfortunately, due to space limitations we could not add further details about this experiment to the paper.

¹⁰ <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html#sklearn.cluster.AgglomerativeClustering>

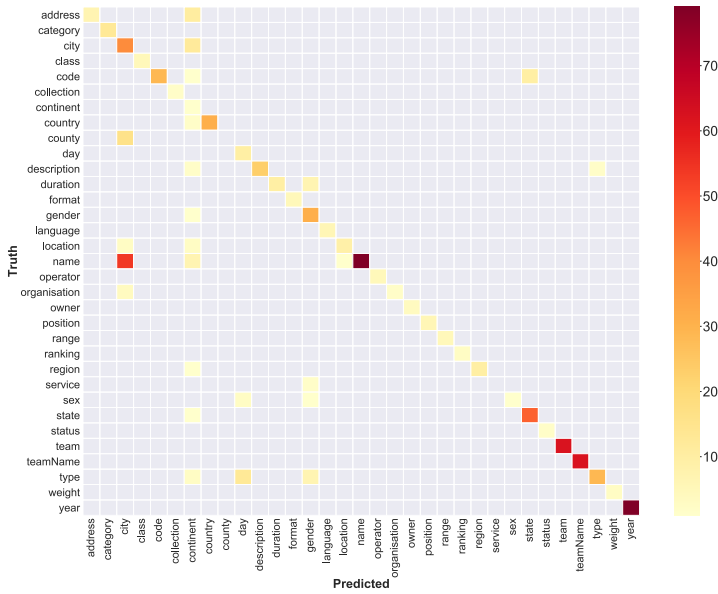


Fig. 4: Confusion matrix of the clustering method

4.3 Future Directions

As mentioned before, in this paper we showed only a very first prototype where we apply the idea of weak supervision for synthesizing labeled training data for semantic type extraction. In our first prototype, we only covered the labeling approach based on clustering as discussed before. Hence, the main avenue of future work is to extend this prototype and add a much broader set of labeling functions.

Furthermore, another direction is to study alternatives for the training data generation process. Currently, we directly use the potentially noisy training data generated by the labeling functions for re-training. Another possible direction as shown in [Ra17], would be to first train a generative model that can learn how to generalize from the additional training data and thus mitigate the negative effects such as noisy data to a certain extent.

Finally, in the current state, we only consider semantic types whose data values are strings or types that provide a semantic meaning when converted to a string (such as *weights* and *dates*). The semantic type detection of numeric types such as *temperature* require additional labeling functions and therefore represent future research.

5 End-to-End Evaluation

In the section before, we have already shown that the basic idea of weak supervision can help to generate training data by clustering to improve the robustness w.r.t different data characteristics. In the following, we report on the initial results of using this approach in an

	Macro average F_1	Precision	Recall	Support-weighted F_1
SHERLOCK (not re-trained)	0.114	0.375	0.309	0.322
SHERLOCK (re-trained)	0.806	0.879	0.859	0.860
SATO (not re-trained)	0.090	0.322	0.304	0.300
SATO (re-trained)	0.811	0.912	0.894	0.894

Tab. 1: Performance comparison of the models on Public BI Benchmark

end-to-end evaluation to show how this can boost the performance of learned type extraction approaches such as Sato.

Setup and Data Preparation: We implemented our approach for automatic labeling in Python using the Google USE embeddings as mentioned before. Moreover, for training and evaluation, we used the source code provided by Sato¹¹. However, Sato is designed to be built and trained from scratch. Hence, we extended Sato with the appropriate functionality for incremental re-training.

End-to-End Results: For showing the end-to-end performance of our approach, we restricted ourselves to the 10% of the Public BI Benchmark data that is supported by Sato and its semantic types. For this, we first generated additional training data and then re-trained the pre-trained Sato model with our additionally labeled data. For generating additional training data, we used the clustering approach discussed before for the Public BI Benchmark. For this purpose, we split the Public BI Benchmark into a training and testing set.

As we see in Tab. 1, after re-training the Sato model with the synthesized training data of our approach, Sato achieves F_1 scores (macro average and weighted) of 0.811 and 0.89 respectively. This is a significant improvement of almost +0.60 compared to the performance of Sato without re-training. In addition to show that our approach also generalizes to other learned approaches, we furthermore used Sherlock [Hu19] (without and with re-training). As shown in Tab. 1, this leads to a similar performance gain. In summary, these results show that our approach is in principle able to boost the performance of learning-based approaches that have been pre-trained on only a small training data set not covering all data characteristics found in a new unlabeled data set.

6 Conclusions

Detecting semantic types for columns of data sets stored in data lakes results in an enormous benefit building a data catalog to address the data discovery problem. While recent papers have shown initial results for learned approaches that can be used for extracting semantic types, they cannot support many real-world data sets since they only support a limited set of semantic data types as we have shown in our study. To tackle this problem, we suggested a new direction of using weak supervision for generating additional labeled training data and use this for re-training the existing learned model. An initial evaluation of our new direction using our current prototype shows that this approach can lead to huge performance gains.

¹¹ <https://github.com/megagonlabs/sato/tree/master>

References

- [AW20] AWS, A.: AWS Glue Concepts - AWS Glue, <https://docs.aws.amazon.com/glue/latest/dg/components-key-concepts.html>, 2020.
- [Az20] Azure, M.: Common Data Model and Azure Data Lake Storage Gen2 - Common Data Model | Microsoft Docs, 2020.
- [Ca18a] Castro Fernandez, R. et al.: Aurum: A Data Discovery System. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE). Pp. 1001–1012, 2018.
- [Ca18b] Castro Fernandez, R. et al.: Seeping Semantics: Linking Datasets Using Word Embeddings for Data Discovery. In: ICDE '18. Pp. 989–1000, 2018.
- [Di14] Dixon, J.: Data Lakes Revisited, <https://jamesdixon.wordpress.com/2014/09/25/data-lakes-revisited/>, 2014.
- [Ha16] Halevy, A. et al.: Goods: Organizing Google's Datasets. In: SIGMOD '16. ACM, pp. 795–806, 2016.
- [Hu19] Hulsebos, M. et al.: Sherlock: A Deep Learning Approach to Semantic Data Type Detection. In: KDD '19. ACM, pp. 1500–1508, 2019.
- [LM14] Le, Q.; Mikolov, T.: Distributed Representations of Sentences and Documents. In: ICML. 2014.
- [LSC10] Limaye, G. et al.: Annotating and Searching Web Tables Using Entities, Types and Relationships. Proc. VLDB Endow. 3/1–2, pp. 1338–1347, Sept. 2010.
- [Ma17] Mathis, C.: Data Lakes. *Datenbank-Spektrum* 17/3, pp. 289–293, Nov. 2017.
- [Ma19] Mahdavi, M. et al.: Raha: A Configuration-Free Error Detection System. In: SIGMOD '19. ACM, 2019.
- [Mi13] Mikolov, T. et al.: Distributed Representations of Words and Phrases and Their Compositionality. In: NIPS'13. Pp. 3111–3119, 2013.
- [Na19] Nargesian, F. et al.: Data Lake Management: Challenges and Opportunities. Proc. VLDB Endow. 12/12, pp. 1986–1989, 2019.
- [Na20] Nargesian, F. et al.: Organizing Data Lakes for Navigation. In: SIGMOD '20. ACM, pp. 1939–1950, 2020.
- [QHV16] Quix, C. et al.: Metadata Extraction and Management in Data Lakes With GEMMS. *Complex Syst. Informatics Model. Q.* 9/, pp. 67–83, 2016.
- [Ra17] Ratner, A. et al.: Snorkel: Rapid Training Data Creation with Weak Supervision. Proc. VLDB Endow. 11/3, pp. 269–282, Nov. 2017.
- [RZ19] Ravat, F.; Zhao, Y.: Metadata Management for Data Lakes. In: *New Trends in Databases and Information Systems*. Springer, pp. 37–44, 2019.

- [So12] Socher, R. et al.: Semantic compositionality through recursive matrix-vector spaces. In: Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning. Association for Computational Linguistics, pp. 1201–1211, 2012.
- [YH18] Yan, C.; He, Y.: Synthesizing Type-Detection Logic for Rich Semantic Data Types Using Open-Source Code. In: SIGMOD '18. ACM, pp. 35–50, 2018.
- [Zh20] Zhang, D. et al.: Sato: Contextual Semantic Type Detection in Tables. Proc. VLDB Endow. 13/12, pp. 1835–1848, July 2020.

Tracing the History of the Baltic Sea Oxygen Level

Evolution and Provenance for Research Data Management

Tanja Auge,¹ Andreas Heuer²

Abstract: In order to guarantee the reproducibility of research results, large research communities, conferences and journals increasingly demand the provision of original research data. Since this is often not possible or desired, a certain tact and sensitivity is needed. With our method, combining provenance and evolution, we can identify the source tuples necessary for the reconstruction of a query result also in temporal databases. To avoid dirty data caused by the inverse evolution, we introduced the *what*-provenance, which remembers the data types of the source relation.

Keywords: Long-term Data; Schema Evolution; Provenance; Research Data Management; CHASE

1 Introduction

"Death zones at the bottom of the sea", "Death zones are growing rapidly: is the Baltic Sea dying?", "Death zones in the Baltic Sea: the air is getting scarce", such and similar headlines have been appearing in the newspapers again and again in the past years. But what is the meaning of this? These are oxygen-free regions at the bottom of the oceans [Ca14]. In the case of the Baltic Sea the *Leibniz Institute for Baltic Sea Research Warnemünde* (IOW) names three causes, whereof two are based on natural conditions. Even if these death zones cannot be predicted, a certain increasing trend can be seen in the last 100 years. However, in order to judge this objectively, long-term evaluations must be carried out.

Imagine a young scientist who is unsettled about these news and now eagerly tries to verify the statements described in the newspapers. To assess the development of oxygen content over the last 100 years, he is planning an article summarizing the results of all studies conducted so far. He wants to reproduce and reconstruct the study results himself. For this purpose, he contacts, among others, the IOW.

To guarantee the long-term availability of observational data and metadata the IOW provides a sophisticated data management system. It enables easy data search and retrieval to complement international data exchange and provide data products for scientific, political,

¹ University of Rostock, DBIS, Germany, tanja.auge@uni-rostock.de

² University of Rostock, DBIS, Germany, andreas.heuer@uni-rostock.de

For the sake of clarity we point to <https://www.io-warnemuende.de/sauerstoff.html> for the description of the causes and more details.

industrial and public actors . This system consists among others of a metadata catalogue and oceanographic point data.

Like many other institutes and universities, the IOW strives after an integrated, reliable and sustainable data management concept, which allows access according to the *FAIR principles* (Findable, Accessible, Interoperable, Reusable). The principles define "characteristics that contemporary data resources, tools, vocabularies and infrastructures should exhibit to assist discovery and reuse by third-parties" [Wi16]. One goal of FAIR is the reuse of data. To achieve this, data and metadata must be described in detail so that they can be replicated and/or combined in different environments . Our contribution is a new approach (see Fig. 1) to the reproducibility and traceability of published research results. This is particularly useful for changing databases.

The young scientist asks for the original data, for example from 1977. However, the IOW can not provide the original data in the original "format" — the data can be on external data storage devices, punch cards or simply on paper. This would be too costly and time-consuming for the IOW. However, the data are additionally stored in an institute-wide, changing database. We can now use the current materialized view and determine the original data of interest to our researcher.

So let's take a closer look at Fig. 1 summarizing our approach: Let there be a published research result (green diagrams on the computer screen) and its associated evaluation query — a query in the case of structured databases including selection, projection, join as well as simple aggregation queries. The provenance enriched inversion of this evaluation query, the so-called provenance query, (III) provides the source tuples necessary for the reconstruction of the result. Thus we obtain a (minimal) sub-database of the original database (red dotted). In case of changing databases, like the over 100 years developed oxygen database at the IOW, this sub-database must be calculated from the current view, of course. For this purpose we first invert the evolution (I) and then execute the evaluation request (II). The result of this query can now be inverted (III) and the minimal sub-database (red dotted) can be reconstructed with the help of Provenance. After the new evolution (IV) we get the evolved (minimal) sub-database (blue dotted), the basis of our reformulated evaluation query (V).

Our method allows us to keep results reproducible, comparable and robust against (interpretation or evaluation) errors. Instead of storing each database version, we only store a (minimal) sub-database. For very large data sets (petabytes and more), which change frequently, data reduction guaranteed by the sub-database, can reduce costs. In addition, the data itself may be worth protecting. This applies not only to personal data but also to research data. For example, the data of MOSAiC, an international project studying the central Arctic, has been generated with great effort and is currently only available to the collaborators of the project. This will not change until 2023, see MOSAiC Data Policy . Military data

<https://www.io-warnemuende.de/datenportal.html>

<https://www.ruhr-uni-bochum.de/researchdata/de/index.html>

<https://spaces.awi.de/display/DM/MOSAiC+Data+Policy>

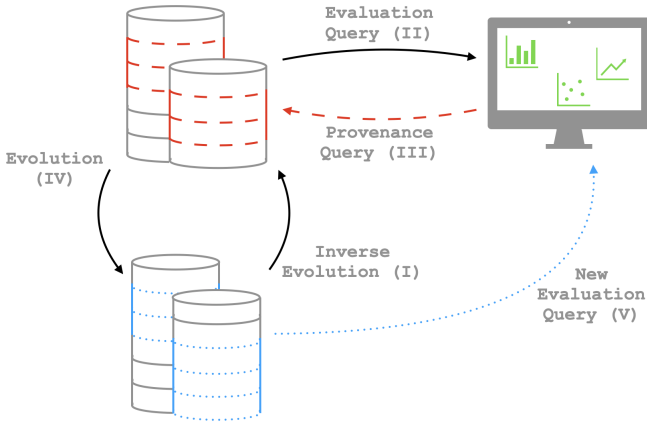


Fig. 1: Combining query evaluation, evolution and provenance: (II + III) The result of an evaluation query (shown as a set of green diagrams on the computer screen) can be inverted using provenance (dashed red). This creates the minimal sub-database top left necessary for the reconstruction of the query result. (I + IV) In case of a temporal databases, the minimal sub-database bottom left must be calculated from the current materialized view. However, this is possible by combining evolution and provenance query. The new evaluation query (dotted blue) results analogously as combination of the inverse evolution and the original query evaluation (V).

have to be protected, too. Even though the data is private, sharing the sub-database still guarantees the traceability of the published research results [Au20a].

Previous provenance queries are usually processed on fixed databases and a specific query. By combining data provenance and evolution we are able to extend provenance queries to temporal databases. First approaches can already be found in [AH18a]. After a short overview of the current status of provenance and evolution in Section 2, we provide a short introduction to IOW (see Section 3). However, the focus of this paper is on the reconstruction of the source tuples in a changing database as well as defining a new provenance type, the so-called *what-provenance* (see Section 4). We will conclude with our future contributions.

2 Provenance and Evolution

Since modern databases do not support schema development proactively, developers often have to intervene manually. However, this is very error-prone and usually not feasible for large data sets. Trying to solve this problem leads to various prototypical implementations: PRISM/PRISM++ [CMZ08] allows to specify evolution steps using so-called *Schema Modification Operations (SMOs)*, defining SMOs as a set of SQL-based schema modification operators including among others Create Table, Add Column and Merge Column. BiDEL [He17] presents SMOs that are relationally complete, invertible and enable forward and

SMOs	SMOs
CREATE Table R(a,b,c)	MERGE Column a,c AS func(a,c) IN R TO d
DROP Table R	SPLIT Column a IN R TO d USING func ₁ , e USING func ₂
ADD Column d [AS const func(a,b,c)] INTO R	
DROP Column c FROM R	
RENAME Column b IN R TO d	

Tab. 1: Schema Modification Operators at IOW [Au20b]

backward query rewriting and data migration, whereas they can guarantee bidirectionality. For this reason a SMO, if it exists, is unique and BiDEL can be considered a nice language to describe SMOs [AK19]. VESEL on the other hand is the first system that allows visual exploration of schema development by means of provenance queries [AK19]. These tools are dependent on versioning the database, thus they have a different notion of provenance.

As shown in [Au20b], all schema evolution steps occurring at the IOW can be represented by SMOs. The language of SMOs consists of eleven operators, which describe the evolution of columns and tables [CMZ08]. In later work these operators were extended by six more, the so-called *Integrity Constraints Modification Operators (ICMOs)* [Cu13]. The SMOs relevant for the IOW, extended by two SMOs for combining and splitting attributes [Au20b] are summarized in Tab. 1.

Data provenance is used to describe the traceability of a query result back to the relevant original data [MH17]. This includes the data set itself (*where*-provenance) as well as the travelled path (*why*- and *how*-provenance). While the *why*-provenance [BKT01] specifies a witness base that identifies the tuples involved in the query result, the *how*-provenance uses provenance polynomials for specify a calculation rule [GT17].

We use the CHASE algorithm [Fa11] as a the foundation for evaluation and provenance queries as well as evolution. For the developed concept, introduced in this paper, it shall be not object of the discussion. The CHASE is a procedure that modifies a given database instance I by incorporating a set of dependencies Σ like s-t tgds (*source-to-target tuple-generating dependencies*) and egds (*equality-generating dependencies*). Generalized, s-t tgds create new tuples and tgds/egds clean the database [Be17].

The representation of the evolution as SMO and thus also as s-t tgd [CMZ08] allows the processing of the evolution by the CHASE algorithm. Since relational algebraic queries can also be represented by s-t tgds and egds, CHASE is also suitable for processing evaluation queries. Given an instance I , an evaluation query Q and an evolution ε , $\text{chase}_Q(I)$ corresponds to our published query result and $\text{chase}_\varepsilon(I)$ to a database instance of a recent version. The inverse evolution as well as the provenance query are calculated in a second step, the so called *BACKCHASE* [DH13, Me14]. In our approach, the BACKCHASE is nothing else than the CHASE algorithm itself, applied to the result of the CHASE. The only difference is the amount of dependencies to be included.

Before discussing the combination of evolution and provenance (see Section 4), we will briefly introduce the IOW, an example for our research use case.

3 Evolution at a Research Institute

Especially for long-term data, schema and data changes are not exceptional. The same applies to the data of the IOW as a use case. It maintains a number of databases on different Baltic Sea specific topics. One of them is the *IOWDB*, the *Oceanographic Database of IOW*. It contains oceanographic readings and metadata (mainly Baltic Sea) from 1877 to 2020, in total more than 78 million measured samples.

One third of the stored data in the IOWDB is obtained with a so-called *CTD probe*. Primary parameters of this instrument are **C**onductivity (electrical conductivity, which is used to determine salinity), **T**emperature and **D**epth, which is determined by the prevailing pressure. The scientific requirements at IOW have changed continuously over the years, which has been accompanied by significant improvements in instrumentation, data acquisition and processing on board research vessels and data storage on land. The data evaluation from different years are therefore an essential task of data processing [Au20b].

When analyzing the evolution operations occurring at the IOW, in particular the addition of attributes using the `ADD Column` as well as the merging and splitting of attributes have proven to be extremely relevant. Almost 80% of all operations are responsible for adding new attributes. Furthermore, 16% of the operations are merging or splitting attributes. The corresponding SMOs `MERGE Column` and `SPLIT Column` and a detailed analysis of the evolution at IOW can be found in [Au20b].

4 Provenance and Evolution for Research Data Management

After we have determined the schema modifications relevant for the integration of old data sets and specified their operators, we now want to deal with the question, if and how a given query can be executed on other schema versions? Accordingly, to execute the original 1977 query on the current materialized view, we are interested in the related (minimal) sub-database. Sometimes additional information is necessary, too. In Fig. 1 the sub-database from 1977 is highlighted as a dashed red box, the later version is highlighted as a green box. The same coloring can be found in Fig. 2, a more detailed depiction of the figure above.

Imagine the following situation: A young scientist is concerned about newspaper articles on the subject of "Death zones in the Baltic Sea". He would like to form his own opinion. In order to judge the development of oxygen content over the last 100 years, he is planning an article summarizing the results of all studies conducted so far. He wants to reproduce and reconstruct the study results himself, which we would like to support.

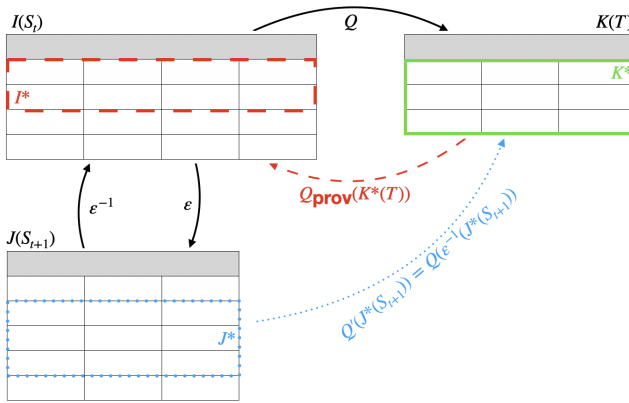


Fig. 2: Unification of query evaluation, provenance and evolution, based on [AH18a]

The IOW's research data is stored persistently for each research project as well as in a developing institute database. For the IOW evaluations, however, we only have one materialized view at our disposal (see $J(S_{t+1})$ in Fig. 2). Since we cannot hand over the entire database $J(S_{t+1})$ — note the case of protected data —, we are looking especially for the data J^* necessary to reconstruct the original result. This data we can submit to our young scientist for his replication study.

So we have to recalculate the old database $I(S_t)$ from our materialized view $J(S_{t+1})$ to execute the original query Q again which was executed some years ago. The repeated execution of Q is necessary to get additional information for the inversion query Q_{prov} . The reconstructed source tuples I^* are then transformed back into the materialized view J^* . On J^* our young scientist can now place all queries that he is interested in. In order to get the same query result K^* on J^* , we have to transform the query Q as well.

We have explained in [AH18a] how this process can look like. The idea we refined and extended to four major steps, of which step **I.** and **II.** can be neglected if the provenance of the query Q is already known.

- I.** Reconstructing the original database by inverting the evolution
- II.** Calculating the query result
- III.** Determining the source tuples using the query result as well as data provenance
- IV.** Evolving the minimal sub-database and transforming the evaluation query

Fig. 2 describes this process in a graphical way. Starting with the current materialized view $J(S_{t+1})$ the inversion ε^{-1} of the evolution ε returns the original database instance $I(S_t)$ and the known query Q the result $K(T)$, highlighted in green. In research data management, K^* always corresponds to the entire result database K , i.e. $K^* = K$, since the complete result of

the scientific evaluation has to be reproducible. Deploying Q_{prov} we determine the minimal sub-database I^* (red dashed box), i.e. the source tuples involved in the result. Finally, ε is used to transform this into the current database version J^* (blue dotted box). But let's have a closer look at the individual steps:

I. Reconstructing the original database by inverting the evolution By using the inverse evolution ε^{-1} the old, original database $I(S_t)$ can be calculated from the current materialized view $J(S_{t+1})$. Thus we obtain: $I(S_t) = \varepsilon^{-1}(J(S_{t+1}))$. Therefore the evolution ε and its (exact) inverse ε^{-1} are formulated as s-t tgds and egds and processed by the CHASE or BACKCHASE [Fal11]. All new tuples, which did not exist in the original version (recognizable by their ID), are deleted and the remaining attributes are processed using the inverse evolution ε^{-1} . Operations like creating a new relation, renaming or inserting new attributes are easy to invert and can be done without further loss of information.

In contrast the inversion of merge- and split-operations, which are a composition of add and drop operators, requires a bit more work. First the used auxiliary function f needs to be invertible or at least quasi-invertible and second the possible data-type-changes need to be considered. For example, merging the attributes DAY, MONTH and YEAR to a common date can be inverted by splitting the date again. Furthermore, this split must ensure that the original attribute value is restored correctly. Since the mathematical inverse and the implemented inverse can differ, there can occur dirty data. To solve the problem of dirty data, we defined the so called *what-provenance* below.

The multiplication of two attribute values can not be inverted without further considerations, as there is no information about the multipliers. The usage of provenance-polynomials [GT17] extended by the possibility to process functions would guarantee the unambiguous inversion of the multiplication too. But without the concrete specification of the attribute values, they will be lost. To the best of our knowledge, such an extension does not yet exist, but we are already working on a solution.

II. Calculating the query result The evaluation query Q can be formulated as s-t tgd or egd and processed by the CHASE algorithm as well. The resulting database instance $K^*(T) = Q(I(S_t))$ is highlighted in green in Fig. 2.

III. Determining the source tuples using the query result as well as data provenance

The result of the evaluation query Q can be calculated with the CHASE algorithm. The subsequent construction of the minimal sub-database I^* is achieved by inverting the query Q . This inverse Q_{prov} returns using the BACKCHASE-algorithm the minimal subdatabase $I^*(S_t) = Q_{\text{prov}}(K^*(T))$, necessary for the calculation of the query result (Fig. 2, red dashed). Here, too, data provenance, especially *how*-provenance, plays a decisive role. Most evaluation queries, apart from the selection for inequality and difference [AH18b, ADT11] can be

inverted by using provenance. By specifying the corresponding CHASE inverse function [Fa11] we can indicate how well the original database I can be reconstructed. An overview of the most common evaluation requests can be found in [AH18b]. The evaluation of the (minimal) sub-database returns the original query result. How closely the sub-database matches the original database depends on the inverse type and provenance query.

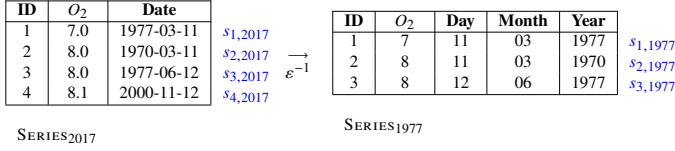
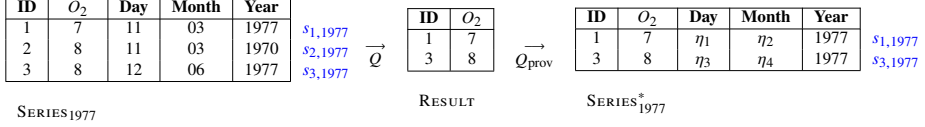
IV. Evolving the minimal sub-database and transforming the evaluation query After determining the minimal sub-database I^* it is transformed into the current materialized view by applying evolution. Here again, CHASE is used. A new minimal sub-database $J^*(S_{t+1}) = \varepsilon(Q_{\text{prov}}(K(T)))$ (Fig. 2, blue dotted) is created, which we make available to the young scientist. Since $Q'(J^*(S_{t+1})) = Q(I(S_t))$, the original query result K^* can now be reconstructed without errors from the minimal materialized view J^* . The query Q' corresponds to the composition of the original evaluation query Q and the inverse evolution ε^{-1} , i.e. $Q'(J^*(S_{t+1})) = Q(\varepsilon^{-1}(S_{t+1}))$.

In summary: We can provide the young scientist with all necessary information under the mentioned conditions (i.e. considering the invertibility of evolution and evaluation). Thanks to the minimal sub-database of our materialized view J^* and the transformed query Q' , he is now able to reconstruct the original published research results. The degree of accuracy depends on the accuracy of the corresponding inverse Q_{prov} and ε^{-1} . So, our young scientist will start his development of the oxygen content of the Baltic Sea with a simple example. Let us examine the evolution of the attribute date in 1977 and 2017. For the sake of simplicity, we will limit the relation series to the three ID, O_2 and date.

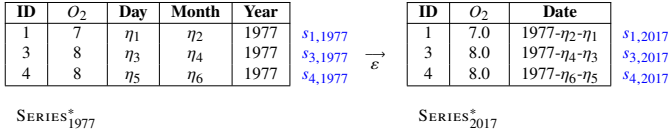
Example 4.1 *We are interested in the progression of the oxygen level O_2 in 1977. Unfortunately the format of the date has changed over the years. While it consisted of three separate attribute values of type `VARCHAR` in 1977, this is now stored in a common date of type `DATE`. In order to determine the source tuples of the relation SERIES_{2017} necessary to detect the oxygen level in 1977, we need the materialized view $J(S_{t+1}) = \text{SERIES}_{2017}$, the evaluation query $Q = \pi_{\text{ID}, O_2}(\text{SERIES}_{1977})$ and the evolution*

$$\varepsilon = \text{MERGE Column DATE AS func (YEAR, MONTH, DAY) INTO SERIES}_{2017}.$$

First we invert the materialized view SERIES_{2017} and obtain the original database instance SERIES_{1977} (Tab. 2a). Then we execute the evaluation query Q and invert the result using data provenance (Tab. 2b). The so generated minimal sub-database SERIES_{1977}^ can now be transformed back to the current materialized view (Tab. 2c). Due to the projection on O_2 as well as the ID all information about the attributes Day and Month is lost. Thus, we insert a null value here. Only the value of the Year can be reconstructed from the blue annotation $s_{i,1977}$. We get SERIES_{2017}^* and the source tuples with the IDs 1, 3 and 4, which are necessary for the reconstruction of the query Q . ■*

SERIES₂₀₁₇SERIES₁₉₇₇(a) Inverse evolution ε^{-1} SERIES₁₉₇₇

RESULT

SERIES₁₉₇₇*(b) Query Q and provenance query Q_{prov} SERIES₁₉₇₇*SERIES₂₀₁₇*(c) Evolution ε

Tab. 2: Calculation of a (minimal) sub-database given the query $Q = \pi_{\text{ID}, O_2}(\text{SERIES}_{1977})$ and the result RESULT

What-Provenance Data provenance describes where a piece of data comes from, why and how it was created. Data provenance currently does not provide any information about how attribute values are defined; in fact, it doesn't provide any information about the allowed data types and their formats [Ma20]. This is particularly important when converting attribute values. This avoids rounding errors or enables using pre-defined SQL functions like Concat or Substring in the merge and split variants introduced in [Au20b]. The fact, that the mathematical inverse of a function might not necessarily match its implemented inverse makes the relevance of such an additional provenance more valid. For example, the data may contain hidden information such as a leading 0 or 1 to identify original or validated data. Also the significant digit is important, i.e. for rounding error calculations. In the case of the oxygen level the values vary between 7 and 9, so a significant digit is important for small changes. In the case of evolution, such changes can occur accordingly (see O_2 in Tab. 2). This is where *what*-provenance comes into play.

Definition 4.1 (*what-provenance*) Let $S_t(A_1, \dots, A_n)$ and $S_{t+1}(A'_1, \dots, A'_m)$ be two temporal versions of the same database. Let ε be the evolution between S_t and S_{t+1} . The *what base* is defined as a set of tuples $(A'_i, (A_k, \mathbb{D}(A_k)) \times \dots \times (A_l, \mathbb{D}(A_l)))$ with $\varepsilon^{-1}(A'_i) = (A_k, \dots, A_l)$ and $\mathbb{D}(A_i)$ domain of the attribute A_i . If A doesn't have a pre-image, we write $(A, (\emptyset, \emptyset))$.

Hence the *what*-provenance provides the attribute itself as well as the associated source data types, the provenance of a certain tuple corresponds to the provenance of the whole relation.

Example 4.2 *We look again at the relation `SERIES` in the years 1977 and 2017. The evolution of O_2 is limited to data type and the evolution of `Date` can be described by the `SMO MERGE Column`. The corresponding *what*-provenance is then:*

$$\{(ID, (ID, \text{INTEGER})), (O_2, (O_2, \text{DOUBLE})), \\ (Year, (Date, \text{DATE})), (Month, (Date, \text{DATE})), (Day, (Date, \text{DATE}))\}.$$

*And the *what*-provenance of the inverse evolution is:*

$$\{(ID, (ID, \text{INTEGER})), (O_2, (O_2, \text{DOUBLE})), \\ (Date, (Year, \text{CHAR}(2)) \times (Month, \text{CHAR}(2)) \times (Day, \text{CHAR}(2))))\}.$$

■

5 Conclusion and Future Work

To guarantee the reproducibility of research results, large communities, conferences and journals increasingly demand the provision of original research data. Since this is often not possible or desired, a certain tact and sensitivity is needed. With our method, the source tuples necessary for reconstruction can be determined to a minimal extent, which is helpful for our young scientist, too. For this purpose we combine the original source tuples reconstructed with data provenance with the evolution of the temporary database. Both, a forward and a backward evolution is possible, depending on which database instance is currently materialized. Additionally, to reduce dirty data caused by the inverse evolution, we introduced the *what*-provenance, which remembers the data types of the source relation.

The detailed analysis of our concept is currently in progress. As stated in Section 2, evolution and evaluation can be formalized using the CHASE. For the inversion we consider five *CHASE-inverse* types [AH18b]. We already examined the inverse types of the evaluation queries [AH18b], for evolution these types still must be identified. Inverse evolution may not necessarily be unique depending on the formalization of the SMO. We also investigate when a specific inverse is useful. There is a natural conflict of interest between publishing original data (provenance) and protecting these data (privacy) introduced in [ASH20].

Acknowledgment

Thanks to the IOW for providing their research data, Erik Manthey for analyzing these data as well as Boris Glavic and Bertram Ludäscher for the exciting discussions. Likewise many thanks to the three reviewers and Tom Ettrich for helpful comments writing this paper.

Bibliography

- [ADT11] Amsterdamer, Yael; Deutch, Daniel; Tannen, Val: Provenance for Aggregate Queries. In: PODS. ACM, pp. 153–164, 2011.
- [AH18a] Auge, Tanja; Heuer, Andreas: Combining Provenance Management and Schema Evolution. In: IPAW. volume 11017 of LNCS. Springer, pp. 222–225, 2018.
- [AH18b] Auge, Tanja; Heuer, Andreas: The Theory behind Minimizing Research Data — Result equivalent CHASE-inverse Mappings. In: LWDA. volume 2191 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 1–12, 2018.
- [AK19] Athinaïou, Christos; Kondylakis, Haridimos: VESEL: Visual Exploration of Schema Evolution using Provenance Queries. In: EDBT/ICDT Workshops. volume 2322 of CEUR Workshop Proceedings. CEUR-WS.org, 2019.
- [ASH20] Auge, Tanja; Scharlau, Nic; Heuer, Andreas: Privacy Aspects of Provenance Queries. <https://arxiv.org/abs/2101.04432>, 2020. [Accepted for ProvenanceWeek 2020].
- [Au20a] Auge, Tanja: Extended Provenance Management for Data Science Applications. In: PhD@VLDB. volume 2652 of CEUR Workshop Proceedings. CEUR-WS.org, 2020.
- [Au20b] Auge, Tanja; Manthey, Erik; Jügensmann, Susanne; Feistel, Susanne; Heuer, Andreas: Schema Evolution and Reproducibility of Long-term Hydrographic Data Sets at the IOW. In: LWDA. volume 2738 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 258–269, 2020.
- [Be17] Benedikt, Michael; Konstantinidis, George; Mecca, Giansalvatore; Motik, Boris; Papotti, Paolo; Santoro, Donatello; Tsamoura, Efthymia: Benchmarking the Chase. In: PODS. ACM, pp. 37–52, 2017.
- [BKT01] Buneman, Peter; Khanna, Sanjeev; Tan, Wang Chiew: Why and Where: A Characterization of Data Provenance. In: ICDT. volume 1973. Springer, pp. 316–330, 2001.
- [Ca14] Carstensena, Jacob; Andersena, Jesper H.; Gustafssonb, Bo G.; Conley, Daniel J.: De-oxygenation of the Baltic Sea during the last century. *J. Artif. Soc. Soc. Simul.*, 111(15), 2014.
- [CMZ08] Curino, Carlo A.; Moon, Hyun J.; Zaniolo, Carlo: Graceful Database Schema Evolution: the PRISM Workbench. *Proc. VLDB Endow.*, 1(1):761–772, 2008.
- [Cu13] Curino, Carlo; Moon, Hyun Jin; Deutsch, Alin; Zaniolo, Carlo: Automating the database schema evolution process. *VLDB J.*, 22(1):73–98, 2013.
- [DH13] Deutsch, Alin; Hull, Richard: Provenance-Directed Chase&Backchase. In: In Search of Elegance in the Theory and Practice of Computation. volume 8000 of Lecture Notes in Computer Science. Springer, pp. 227–236, 2013.
- [Fa11] Fagin, Ronald; Kolaitis, Phokion G.; Popa, Lucian; Tan, Wang Chiew: Schema Mapping Evolution Through Composition and Inversion. In: Schema Matching and Mapping, Data-Centric Systems and Applications, pp. 191–222. Springer, 2011.
- [GT17] Green, Todd J.; Tannen, Val: The Semiring Framework for Database Provenance. In: PODS. ACM, pp. 93–99, 2017.

- [He17] Herrmann, Kai; Voigt, Hannes; Behrend, Andreas; Rausch, Jonas; Lehner, Wolfgang: Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In: SIGMOD Conference. ACM, pp. 1101–1116, 2017.
- [Ma20] Manthey, Erik: , Beschreibung der Veränderungen von Schemata und Daten am IOW mit Schema-Evolutions-Operatoren. Bachelor Thesis, University of Rostock, DBIS, 2020.
- [Me14] Meier, Michael: The backchase revisited. VLDB J., 23(3):495–516, 2014.
- [MH17] Melanie Herschel, Ralf Diestelkämper, Housseem Ben Lahmar: A survey on provenance: What for? What form? What from? VLDB J., 26(6):881–906, 2017.
- [Wi16] Wilkinson, Mark D. et al.: The FAIR Guiding Principles for scientific data management and stewardship. Scientific Data, 3(1):160018, 2016.

(Industrial) Use Cases & Applications

The Data Lake Architecture Framework: A Foundation for Building a Comprehensive Data Lake Architecture

Corinna Giebler,¹ Christoph Gröger,² Eva Hoos,² Rebecca Eichler,¹ Holger Schwarz,¹
Bernhard Mitschang¹

Abstract: During recent years, data lakes emerged as a way to manage large amounts of heterogeneous data for modern data analytics. Although various work on individual aspects of data lakes exists, there is no comprehensive data lake architecture yet. Concepts that describe themselves as a “data lake architecture” are only partial. In this work, we introduce the data lake architecture framework. It supports the definition of data lake architectures by defining nine architectural aspects, i.e., perspectives on a data lake, such as data storage or data modeling, and by exploring the interdependencies between these aspects. The included methodology helps to choose appropriate concepts to instantiate each aspect. To evaluate the framework, we use it to configure an exemplary data lake architecture for a real-world data lake implementation. This final assessment shows that our framework provides comprehensive guidance in the configuration of a data lake architecture.

Keywords: Data Lake; Data Lake Architecture; Framework

1 Introduction

In recent years, data lakes emerged as platforms for big data management and analyses [Ma17b]. They are used in various domains, e.g., healthcare [RZ19] or air traffic [Ma17a], and enable organizations to explore the value of data using advanced analytics such as machine learning [Ma17b]. To this end, data of heterogeneous structure are stored in their raw format to allow analysis without predefined use cases.

However, implementing a data lake in practice proves challenging, as no comprehensive data lake architecture exists so far. Such an architecture specifies, e.g., the data storage or data modeling to be used. In this work, we define *comprehensive* as “*all necessary architectural aspects of a data lake and their interdependencies are covered*”. An architectural aspect is a perspective on a data lake architecture, such as data modeling or infrastructure. To define a comprehensive data lake architecture, multiple such aspects have to be considered. While some concepts are called “data lake (*reference*) architecture” (e.g., in [Sh18]) by their authors, they only focus on a subset of necessary architectural aspects.

In addition, the possible applications of data lakes are very diverse. A data lake might only process batch data [Ma17a] or both batch data and data streams [MM18]. It might be limited

¹ Universität Stuttgart, IPVS, 70569 Stuttgart, Germany {firstname.lastname}@ipvs.uni-stuttgart.de

² Robert Bosch GmbH, 70469 Stuttgart, Germany {firstname.lastname}@de.bosch.com

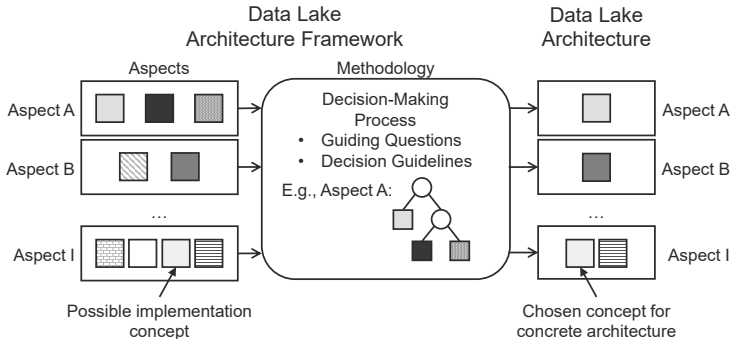


Fig. 1: The data lake architecture framework contains possible implementation concepts. To configure a data lake architecture, concrete concepts are chosen from the framework using the contained methodology.

to data scientists and advanced analytics [ML16] or additionally support traditional data warehousing [Ma17b]. Depending on the kind of data in the data lake and on the scenario in which it is used, different requirements are posed on a data lake architecture. Thus, defining a generic and universally applicable data lake architecture proves difficult. Instead, we propose the data lake architecture framework (DLAF) as a foundation for comprehensive data lake architecture development. Architecture frameworks exist in various domains, e.g., the Zachman framework [Za87] provides both guidance and a methodology to define an appropriate information system architecture. However, in the context of data lakes, we are not aware of such an approach. Fig. 1 depicts the connection between the framework and a configured data lake architecture. The guidance provided by the DLAF is threefold: 1) It defines the architectural aspects necessary for a data lake, e.g., data modeling, 2) it associates each aspect with a set of possible implementation concepts, e.g., data vault [Li12], and 3) it provides a methodology that helps picking appropriate concepts for a data lake architecture while also considering interdependencies between aspects, e.g., between data modeling and infrastructure. A data lake architecture derived from the framework can be understood as a DLAF instance. In this paper, we make the following contributions:

- From a categorization of literature on data lakes, we identify their necessary architectural aspects. We use these aspects to build the data lake architecture framework, which serves as a support for the configuration of a comprehensive data lake architecture.
- We present a methodology as part of the data lake architecture framework that guides the development of a specific data lake architecture from the aspects in the framework.
- We assess the data lake architecture framework with regards to its comprehensiveness and applicability. This assessment shows that the data lake architecture framework is not missing any important aspects for data lakes, and that its methodology is applicable in practice to configure a comprehensive data lake architecture.

- We show how the framework can be used to configure a comprehensive data lake architecture, to evaluate existing data lake architectures, and to extend incomplete data lake architectures to become comprehensive.

The remainder of this work is structured as follows: Sect. 2 discusses related work on data lake architectures and architecture frameworks. Sect. 3 describes the developed DLAF, before Sect. 4 presents the contained methodology for its use. Sect. 5 assesses the framework by analyzing existing data lake implementations and by defining an exemplary comprehensive data lake architecture for real-world industry. Finally, Sect. 6 concludes the work.

2 Related Work

In literature, multiple so called “data lake architectures” are proposed (e.g., in [In16; JQ17; RZ19; Sh18]). The goal of these architectures is to be generic. Sawadogo and Darmont differentiate three kinds of data lake architectures [SD20]: 1) *functional architectures* that cover data ingestion and storage, e.g., [JQ17], 2) *data maturity-based architectures*, where data are organized according to their refinement level, e.g., [Sh18], and 3) *hybrid architectures* that combine both. As functional architectures and data maturity-based architectures focus only on singular aspects of the data lake [SD20], they do not qualify as comprehensive data lake architectures. We thus only consider hybrid architectures in the remainder of this section. Sawadogo and Darmont name two hybrid architectures: Inmon’s data pond architecture [In16] and Ravat’s data lake functional architecture [RZ19]. To the best of our knowledge, no other generic hybrid architectures are available.

However, defining data ingestion, data storage, and data organization is not sufficient for a data lake. Examples for further aspects of importance are data modeling and metadata management [Gi19a]. Neither Inmon’s nor Ravat’s architecture cover these additional aspects. While there is work on both data modeling and metadata management in data lakes (e.g., [Ei20; HGQ16; Ho17; NRD18]), it focuses only on singular aspects. Overall, none of the generic data lake architectures in literature is comprehensive.

In addition to the generic architectures, there are hybrid data lake architectures in specific implementations, e.g., in [Ma17a; MM18]. These architectures are however tailored to specific use cases and are not applicable as generic data lake architectures. Thus, to the best of our knowledge, there is no guidance for defining a comprehensive data lake architecture.

3 Aspects forming the Data Lake Architecture Framework

To address the lack of support for configuring a comprehensive data lake architecture, we present the data lake architecture framework (DLAF) as a foundation for such a configuration. The framework consists of two parts: I) It describes necessary architectural aspects of a

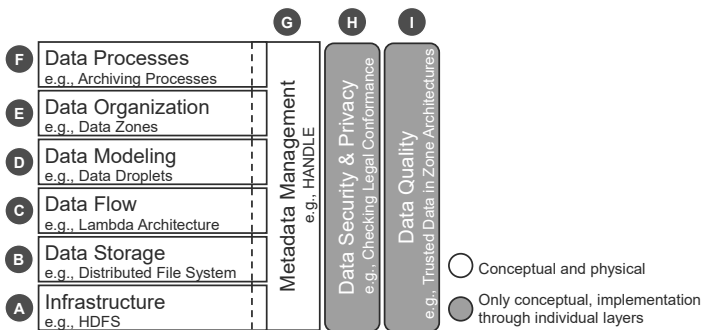


Fig. 2: The data lake architecture framework consists of nine data lake aspects that have to be considered when creating a comprehensive data lake architecture.

data lake. In doing so, it defines the scope for a comprehensive data lake architecture. This section first describes the DLAF aspects that represent the different architectural aspects (Sect. 3.1) before detailing on their interdependencies (Sect. 3.2). II) Moreover, the DLAF includes a methodology to configure a comprehensive data lake architecture, guiding the selection of appropriate concepts for each aspect. This methodology is presented in Sect. 4.

Each part of the framework in Fig. 2 represents one architectural aspect of data lakes associated with a set of concepts for its implementation (cf. Fig. 1). The architectural aspects included in the DLAF were derived by clustering the results of a thorough literature review on concepts for data lake implementation (cf. [Gi19a]). The nine resulting clusters we formulated into disjoint architectural aspects depicted in Fig. 2: A) infrastructure, B) data storage, C) data flow, D) data modeling, E) data organization, F) data processes, G) metadata management, H) data security & privacy, and I) data quality. These aspects can neither be combined further, as they describe different perspectives on the data lake, nor did we find further aspects to be considered. Aspects A-F are sorted by their abstraction degree, with infrastructure as the most physical aspect at the bottom and data processes as the most abstract aspect at the top. Aspects G-I span all of these aspects. We differentiate between aspects that consist of a concept and its physical implementation (white), and aspects that comprise only a conceptual view but are implemented through other aspects (grey). For example, if the data security & privacy aspect requires data encryption, the infrastructure has to offer this functionality. The following paragraphs explain the architectural aspects.

3.1 DLAF Aspects

A) Infrastructure. The infrastructure aspect comprises concepts for the physical implementation of the data lake. The focus lies on concrete storage systems and tools, e.g.,

HDFS³ as distributed file system or MySQL⁴ as a relational database, and their deployment on-premise or in the cloud. An example is given by [Zi15], who use *Hadoop*⁵ and *DB2*⁶. As an exemplary concept for deployment, *hybrid data lakes* [Lo16] are data lakes built both on-premise and in the cloud.

B) Data Storage. The data storage focuses on the types of systems and tools used for data storage and processing (e.g., *file systems* and *NoSQL databases*, or *batch processing* and *stream processing tools*). In contrast to the infrastructure aspect, no specific tools are chosen. Exemplary data lakes are built on a *single distributed file system* (e.g., [Ma17a]) or on *multiple storage systems* (e.g., [Zi15]).

C) Data Flow. The data flow aspect covers the architecture and interaction for the two modes of data movement that may occur in a data lake: batch data and streaming data. Batch data are persistently stored in a storage system and are processed in large volumes [CY15]. In contrast, streaming (or real-time) data are continuously delivered into the data lake and typically need to be processed immediately [CY15]. Streaming data can also become batch data if it is stored for later use. Examples for data flow concepts are hybrid processing architectures such as the *Lambda architecture* [MW15] or *BRAID* [Gi18].

D) Data Modeling. The data modeling aspect describe whether and how data are modeled within the data lake. Typically, the modeling technique used will differ depending on the data's characteristics and their usage. Examples of data modeling techniques applicable in data lakes are *data droplets* [Ho17] or *data vault* [Li12].

E) Data Organization. The data organization aspect defines the conceptual set-up within the data lake. To this end, associated concepts describe what data can be found where and what state they are in (e.g., raw or pre-processed). Examples are the *data pond architecture* [In16], the *zone architectures* (e.g., [Gi20; Sh18]), *Jarke's and Quix' conceptual architecture* [JQ17], and *data meshes* for semantical data organization [De19].

F) Data Processes. The data processes aspect comprises all concepts that focus on data movement and processing. Data processes can be divided into processes for *data lifecycle management* and for *data pipelining*. *Data lifecycle management processes* manage the data from creation and obtaining to disposure [DA17]. These processes have to be carefully defined and standardized within a data lake to facilitate self-service, ensure data usability, and support legal compliance. In contrast, *data pipelining processes* focus on the technical ingestion, movement, and processing of data, such as extract-transform-load (ETL) processes. They are used to describe, e.g., how data move between zones in a zone model. In contrast to the data flow aspect, the data processes aspect rather describes what is done with data instead of focusing on the characteristics of the data themselves. An exemplary data lifecycle management process for data lakes is the *archiving process* given in [Ch15]. Examples for

³ hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

⁴ www.mysql.com/de/

⁵ hadoop.apache.org/

⁶ www.ibm.com/analytics/db2

data pipelining can be found in most zone architectures (e.g., in [Gi20]), or in Jarke's and Quix' conceptual data lake architecture [JQ17].

G) Metadata Management. The metadata management aspect comprises two sub-aspects: The first one, *metadata as enabler*, overlaps the horizontal aspects in Fig. 2. In this sub-aspect, metadata enable other aspects. For example, metadata describe what zone a piece of data belongs to in a zone model or when data was created for lifecycle management. The other sub-aspect is *metadata as a feature*, depicted as the right side of the metadata management aspect in Fig. 2. This sub-aspect contains the functionalities that metadata can provide in addition to their enabler capabilities, such as business glossaries [Ba14] or semantical descriptions. For all metadata, aspects A-F have also to be defined, as metadata have to be stored, modeled, etc. Exemplary concepts are metadata management systems, such as *Constance* [HGQ16], or metadata models, such as *HANDLE* [Ei20].

H) Data Security & Privacy. Data security & privacy is a solely conceptual aspect. It is of great importance in a data lake, as it ensures legal conformance, alignment with business objectives, and much more [Ch15]. Exemplary concepts for this aspect are, e.g., *checking legal conformance* in the data wrangling process [Te15] or *AMNESIA* for GDPR-compliant machine learning [St20], where data security & privacy are implemented through zones.

I) Data Quality. The data quality aspect is also solely conceptual. Maintaining data quality is important to ensure the data's usability and prevent the data lake from turning into a data swamp [Ch14]. While there are *data quality tools*, e.g., Informatica data quality⁷, these still rely on implementations of other aspects, e.g., metadata management. Data quality can also be found in, e.g., data organization, where some *zones hold trusted data* (e.g., [Gi20]).

3.2 Interdependencies between DLAF Aspects

The aspects of the DLAF are not independent from each other, as decisions for one aspect affect other aspects. For example, the aspect of data storage influences data modeling, as different kinds of storage systems support different kinds of modeling (e.g., relational modeling for relational databases, graphs for graph databases). This section explores the interdependencies between aspects and their implications. Fig. 3 depicts these interdependencies as a graph. They are in line with interdependencies as described in existing literature. In this graph, six of the overall nine aspects are visually grouped together for a simpler visualization. The influences between aspects depicted in this graph form the basis of the methodology presented in Sect. 4.

The graph shows that the aspects *data security & privacy* and *data quality* influence all other aspects of the DLAF (I1-I4). This is because these two aspects are not implemented directly, but instead implemented through other aspects. Thus, decisions made for data security & privacy and data quality have to be considered when choosing concepts for

⁷ www.informatica.com/de/products/data-quality/informatica-data-quality.html

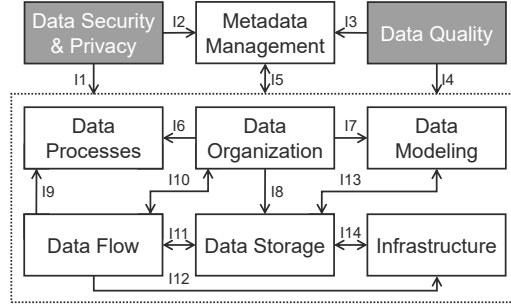


Fig. 3: The influence graph for the DLAF aspects. Directed arrows point from the influencing aspect to the influenced aspect.

all other aspects. For the aspect of *metadata management* all aspects produce and rely on metadata, e.g., metadata describing how data should be processed (I5). They influence what metadata should be collected and how it should be organized. At the same time, metadata are data and thus need to be considered when defining all other aspects (I5).

The *data organization* aspect influences data processes (I6), as data pipelining processes have to be adapted to the chosen data organization, e.g., data zones or ponds. Data organization also influences data modeling (I7), as e.g., data zone architectures typically dictate standardized data modeling in at least one zone. This means that all data in this zone are modeled according to specified rules, e.g., the rules of data vault. Furthermore, data organization influences data storage (I8), as, e.g., the data pond architecture, where data are separated by their structure, necessitates a different storage concept than, e.g., data zone architectures. The *data flow* aspect influences data processes (I9), data organization (I10), data storage (I11), and infrastructure (I12). This is because the data flow aspect comprises the different modes of data (batch and stream). Depending on the decisions made for this aspect, other aspects have to support the respective modes as well. For example, if the concept chosen for the data flow aspect includes stream processing, the concept chosen for infrastructure has to include stream processing engines. However, for data organization and data storage (I10, I11), the influence can also be reversed, as, e.g., using a batch-only data organization or choosing batch and stream processing in data storage dictates a certain data flow. The aspect of *data storage* influences and is influenced by both data modeling (I13) and infrastructure (I14). Depending on the types of storage systems chosen, certain data modeling techniques can or cannot be used. For example, if relational storage is chosen, data models should be applicable to relational schemata. At the same time, the choice of a certain data model will necessitate a different data storage concept. Data storage and infrastructure are closely connected, as one chooses the types of systems while the other defines the concrete systems and tools.

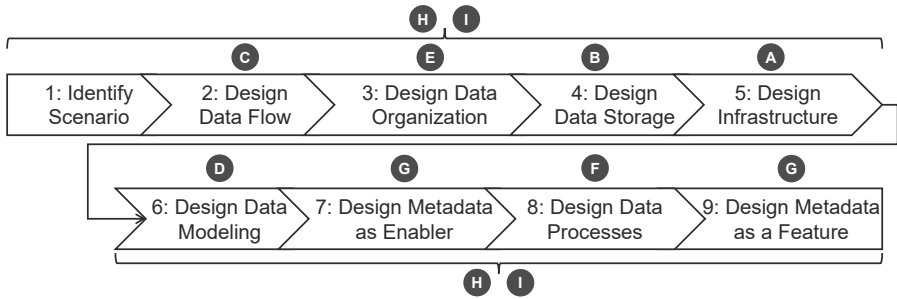


Fig. 4: To configure a comprehensive data lake architecture with the DLAF, nine steps are necessary. Each step is associated with different DLAF aspects, depicted in the circles.

4 Methodology for Configuring a Data Lake Architecture

To configure a comprehensive data lake architecture from the framework, specific concepts have to be chosen from the set of concepts associated with each aspect (see Fig. 2). This section provides a methodology for this task consisting of nine steps. Several of these steps directly correspond to aspects of the DLAF (see Fig. 4), and the order of the steps was determined from the aspects' interdependencies (see Fig. 3). As shown in Fig. 4, the aspects data security & privacy and data quality are associated with steps 1 through 9. This means that during all steps, these two aspects have to be taken into consideration and included accordingly. Our methodology provides guiding questions for each step that support the selection of appropriate concepts for architectural aspects. To the best of our knowledge, no further sources on such questions exists. In addition, we include typical concepts and associated decision guidelines for each aspect. However, these are not exhaustive due to the wide variety of available concepts for implementation.

Step 1: Identify Scenario. Understanding the key requirements of the data lake's application scenario is a crucial prerequisite for all following architecture decisions. To this end, the following questions should be answered: 1) What kind of data are managed in the data lake? What are their characteristics? This question also targets the data security & privacy and data quality requirements. 2) What time requirements are associated with data and their usage (e.g., real-time, hourly updates)? 3) How are data used (e.g., advanced analytics only or also reporting)? Further requirement elicitation should also be performed in this step.

Step 2: Design Data Flow. To determine a suitable data flow concept, especially question 2 from Step 1 (what time requirements are associated with data and their usage) is of relevance. If data should be both processed in real-time and in larger time intervals, both batch and stream processing should be used. Hybrid processing architectures comprise both, such as the Lambda architecture [MW15] or BRAID [Gi18]. A guiding question to this decision is: Are batch and stream processing independent from each other or should results from one be available to the other? If they are independent, the Lambda architecture is a sufficient concept. If data and results should be exchanged, BRAID offers the needed functionality.

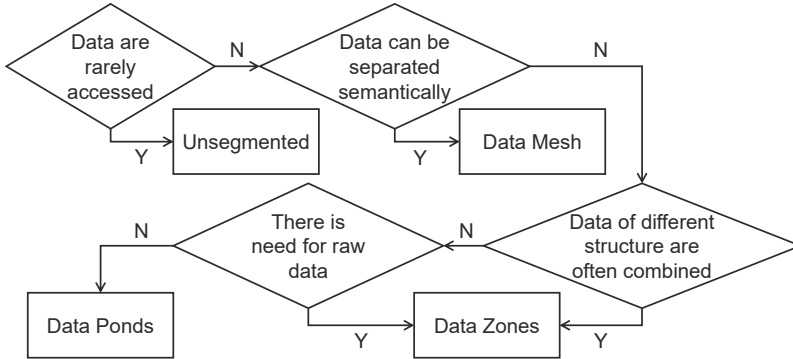


Fig. 5: The decision process for choosing an appropriate data organization concept. Combinations of concepts are not included due to space restrictions.

Step 3: Design Data Organization. Data organization focuses on the efficient management of data for different uses. Thus, to choose an appropriate concept for this aspect, question 3 from step 1 (how are data used) is of high importance again. Fig. 5 depicts a possible decision process for data organization, including some of the guiding questions to be asked. In our example, the concepts to choose from are no segmentation of the data, semantical data meshes [De19], Inmon's data pond architecture [In16], or data zone architectures (e.g., [Gi20]). A properly segmented and structured data lake provides more efficient access and usage than an unsegmented one, however, segmenting the data lake increases its complexity. Note there might be other suitable concepts for data organization, as well as combinations of these concepts. In addition, some of the concepts mentioned require further definition, as there are multiple variants of, e.g., the data zone architecture [Gi19a]. Concepts for data quality are included in most zone architectures and in the pond architecture. Data security & privacy however are only included in some zone architectures (e.g., [Gi20; Go16]) and not in the data pond architecture. For other data organization concepts, neither of the two aspects is considered. However, these aspects cannot be neglected.

Step 4: Design Data Storage. The configuration of a data storage concept depends on the kind of data to be managed (question 1 from step 1) and how they are used (question 3 from step 1). Exemplary guiding questions for this aspect are: Are multiple types of storage systems necessary? Which storage systems can support the data's characteristics? For example, relational databases provide the most appropriate support for structured data. If the managed kinds of data are widely varied, a combination of storage systems might be appropriate. When working with data ponds, each pond can be realized on a different system, e.g., an relational database for the analog data pond and a file system for the textual data pond. For this decision, exemplary guiding questions are: How are data used? What characteristics have to be supported? For example, highly connected data should be managed in a graph database, while structured data can be stored in relational databases. Further decision support can be found in e.g., [Ge17]. It is necessary to consider data security &

privacy and data quality requirements when defining the data storage aspect, as different types of data storage systems support different degrees of consistency, constraints, etc.

Step 5: Design Infrastructure. In this step, the defined data storage and data flow concepts are used to decide on an appropriate infrastructure for the data lake. Storage systems and data processing tools are constantly maturing, and the requirements towards infrastructure are manifold. We thus do not provide details on infrastructure decision support in this paper. However, some guiding questions are: What ingestion rates are required? Are indexes or foreign keys needed? What read/write performance should be offered? Infrastructure can then be chosen in accordance with the answers to these questions.

Step 6: Design Data Modeling. The answers given for question 1 and 3 from step 1 (what data are managed and how are they used) are of great importance for deciding on data modeling concepts, as they determine which data models are suitable. Exemplary guiding questions for this step are: How should structured and semi-structured data be modeled? For example, data vault [Li12] can be used to model these data in data lakes [Gi19b]. How can data be connected across systems? How can unstructured data be connected to structured data? Possible answers to these questions are data droplets [Ho17] or link-based integration [GSM14]. If zones are used as a concept for data organization, modeling concepts differ for each zone. Typically, one zone holds raw data replicated from the source, while another zone contains data in a standardized format, or even in a use-case specific format (e.g., as dimensional schema). Requirements towards data security & privacy and data quality have to be addressed, e.g., through separately treated tables for sensitive data or data models that consolidate data.

Step 7: Design Metadata as Enabler. The leading question to configure metadata as enabler is: What information is needed on the data to manage them meaningfully? This includes 1) metadata that are needed to reflect the concepts chosen in other aspects (e.g., metadata describing the zone of a data), and 2) metadata that are needed for the general operation of the data lake (e.g., information on lineage, access operations, or last-accessed dates). Some metadata are needed for the execution of data processes defined in step 8. Thus, it might be necessary to revisit this step during the definition of data processes to add additional metadata needed. Step 7 also includes metadata for data security & privacy and data quality, such as security classifications, a to-be-deleted date, or known quality issues. As the metadata as enabler identified in this step may vary greatly from one application scenario to another, it is impossible to provide a decisive guideline. Choosing a flexible metadata management model such as HANDLE [Ei20] is beneficial, as it can be adapted and even extended later on. If metadata management has not been considered as data in steps 1-6, step 7 is to fill these gaps. Metadata, just like other data, are in need of infrastructure, data storage, data flow, data modeling, and data organization concepts.

Step 8: Design Data Processes. Due to space restrictions, we cannot provide detailed guidelines for the data process configuration. Some guiding questions for this aspect are: How do data move in the data lake? How are they processed? What is the data's lifecycle?

Most data organization concepts include appropriate data process concepts, e.g., how data move and are processed between zones/ponds in data zone architectures (e.g., [Gi20]) and the data pond architecture [In16]. These processes have to be adapted and extended to fit the concepts chosen for the remaining aspects. If it turns out that data processes require further metadata, step 7 is revisited here. Data processes for data security & privacy, such as processes for accessing sensitive data, and data quality have to be chosen meaningfully to fit the application scenario's needs. The data wrangling process [Te15] and existing lifecycle management processes can serve as a base for the data process configuration.

Step 9: Design Metadata as a Feature. This final step includes all functionality that goes beyond the simple description of data. Metadata management systems such as data catalogs [Ch15] or data marketplaces [Mu13] offer functionalities that go beyond the scope of metadata as enabler, namely semantical data access or data purchase offers. As these additional functionalities can only be implemented with a detailed knowledge on the data lake's architecture, this step is done last. This part of the data lake can be designed quite freely. An associated guiding question is: What further benefit can metadata provide?

5 Assessment and Application of the DLAF

In this section, we assess the DLAF's suitability as architecture configuration guidance in two ways: 1) we analyze existing data lake implementations and sort their architectural decisions into the DLAF's aspects to demonstrate the framework's comprehensiveness (Sect. 5.1). The DLAF aids us in identifying shortcomings of existing architectures and provides pointers for improvement. 2) We assess the methodology's applicability by defining an exemplary data lake architecture using the DLAF (Sect. 5.2).

5.1 Comprehensiveness of DLAF

We use two real-world data lake implementations for the evaluation of the DLAF's comprehensiveness, in particular AIRPORTS DL [Ma17a] and the Smart Grid Big Data Ecosystem [MM18]. We chose these implementations because they provide detailed information on the concepts used and were evaluated using real-world data. They cover two different domains (air traffic, smart grids) and deal with different data management requirements. Neither of these papers includes a methodology for the configuration of their data lake. Tab. 1 matches the decisions made in these implementations with the DLAF aspects. Based on this categorization, we discuss the implementations' comprehensiveness and how they should be extended.

AIRPORTS DL. The AIRPORTS DL [Ma17a] focuses on storing surveillance data of flights, such as a plane's position or altitude. These data are combined with data from third parties, such as weather data, and are streamed into the data lake. The middle column in Tab. 1

DLAF Layer	AIRPORTS DL [Ma17a]	Smart Grid Big Data Eco-System [MM18]
A. Infrastructure	Hadoop (HDFS, MapReduce), Apache Flume, Apache Spark, Apache Oozie, Apache Pig, Apache Atlas, R Studio, Shiny, Apache Sqoop	Hadoop (HDFS, MapReduce), Apache Flume, Apache Spark Streaming, Apache Spark SQL, Apache Hive, Apache Impala, Radoop, Matlab, Tableau, Google Cloud Computing
B. Data Storage	Single File System	Single Eco-System
C. Data Flow	Data are ingested as streams, but processed as batches	Based on the Lambda Architecture, data are processed as stream and as batches
D. Data Modeling	Raw Messages, AIRPORTS Data Model	Undefined
E. Data Organization	Four Zone Architecture	Two Zone Architecture for the data storage (Master data and Serving Layer) based on Lambda Architecture
F. Data Processes	Processing Pipeline for Messages (ETL Processes), Processes for Ingestion and Use	Processing Pipeline from the Lambda Architecture
G. Metadata Management	Managed by Apache Atlas	Undefined
H. Data Security & Privacy	Tracking manipulation of data	Undefined
I. Data Quality	Tracking manipulation of data, Quality through Zones	Undefined

Tab. 1: Categorization of Architectural Decisions in Existing Data Lake Implementations

lists the architectural decisions made in this data lake implementation with respect to the aspects of the DLAF. The following paragraphs detail selected DLAF aspects. There is no explicit explanation for the *data flow* aspect in the paper, aside from data being ingested as data streams. However, data are stored before being processed. In addition, the tools used for processing (e.g., Hadoop MapReduce⁸, Apache Pig⁹) are for batch processing. These decisions suggest that data are processed in batches only and not as data streams. For *data modeling*, data first are stored as raw key-value messages. Then, as they move through processing, they are transformed to fit the AIRPORTS data model, which was specifically created for this application scenario. For *data organization*, a zone architecture consisting of four zones, called layers in the paper, was chosen. These layers are 1) Raw Layer, 2) Alignment Layer, 3) Flight Leg Reconstruction Layer, and 4) Integration Layer. Data are ingested into the Raw Layer and then processed from layer to layer. Finally, data are made available in the Delivery Layer, which is not a processing layer, but an interface to the data lake. The *data processes* in this data lake implementation mostly focus on the movement of

⁸ hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

⁹ pig.apache.org/

data between zones. In addition, it is defined how data are ingested into the data lake and how they can be analyzed and delivered to external systems. Apache Atlas¹⁰ is used to implement a governance system on top of the data lake, which provides *metadata management*, and basic *data security & privacy* and *data quality* by tracking data manipulation. The data quality aspect is also addressed by the zone architecture of the AIRPORTS DL, where the quality of data is increased from one zone to the other.

Overall, all architectural decisions of the AIRPORTS DL can be reflected by the aspects of the DLAF. It also shows that the AIRPORTS DL provided a concept for each of the DLAF aspects. Thus, the AIRPORTS DL architecture covers all aspects from infrastructure to data quality and thus underlines the comprehensiveness criteria.

Smart Grid Big Data Eco-System. The second data lake implementation analyzed is applied in a scenario of smart grids as part of a smart grid big data eco-system [MM18]. The data to be stored and analyzed in this scenario are highly diverse, including sensor data from, e.g., farms and consumers, but also images and videos from plant security cameras. These data are enriched with data from additional sources, e.g., weather data. Data are ingested into the data lake as a stream. The right column of Tab. 1 summarizes the architectural decisions in this data lake. The *data storage* of this data lake is realized as a single system, namely the Hadoop eco-system as seen in *infrastructure*, including HDFS, Hive¹¹, and Impala¹². The *data flow* concept of this data lake is based on the lambda architecture [MW15]: Data ingested as data stream are forwarded to both batch processing, where they are stored persistently, and to stream processing, where they are processed in real-time. The results from both processing modes are combined in a Serving Layer. The usage of the lambda architecture influences the *data organization* concept. According to the lambda architecture, the data lake is divided into two zones: a Raw Zone, where data are persistently stored before processing, and the Serving Layer that holds the processing results. Similarly, *data processes* are given by the lambda architecture.

While all architectural decisions of this implementation could be assigned to a DLAF aspect, this analysis shows that the architecture of the smart grid data lake is not comprehensive. There are no concepts for *data modeling*, *metadata management*, *data security & privacy*, or *data quality*. However, without these concepts, a data lake risks turning into a data swamp, where data are unusable [Ch14]. Using the DLAF, this data lake can be re-designed including extensive metadata management to also address security and quality.

5.2 Application of the DLAF Methodology

In the second part of our assessment, we configure a data lake architecture using the DLAF and its methodology introduced in Fig. 2 and Fig. 4. In doing so, we evaluate the

¹⁰ atlas.apache.org/

¹¹ hive.apache.org/

¹² impala.apache.org/

Step	Resulting Decision
1: Identify Scenario	Structured, semi-structured, and unstructured data Batch and stream processing Both advanced and traditional analyses
2: Data Flow	Hybrid Processing Architecture BRAID
3: Data Organization	Zone Reference Model
4: Data Storage	Multi-Storage System
5: Infrastructure	Hadoop (HDFS, MapReduce), Kafka, MySQL, Apache Spark, . . . , partially Cloud-based
6: Data Modeling	Data Vault, Link-Based Integration
7: Metadata as Enabler	Metadata Types based on HANDLE
8: Data Processes	Organization Specific Processes
9: Metadata as Feature	Data Catalog

Tab. 2: Overview of Resulting Decisions in the Definition of an Exemplary Data Lake Architecture

applicability of the DLAF based on a real-world industry case from a large, globally active manufacturer. The manufacturer’s business is highly diverse, with business domains ranging from manufacturing to quality management to finance. To increase efficiency and reduce costs, an enterprise-wide data lake is implemented to employ data analytics in everyday business. Tab. 2 provides an overview over the decisions made for the data lake architecture. The following paragraphs describe the decision process that led to these solutions. The data flow and data organization of the resulting data lake architecture is depicted in Fig. 6.

Step 1: Identify Scenario. In the use case introduced above, a wide variety of data are used for a multitude of different projects and analyses. To create a proper base for the definition of an exemplary data lake architecture, we refer to the questions defined in Sect. 4, Step 1. Based on these answers, the remaining steps are performed to configure a data lake architecture suited for the manufacturer’s needs.

1) Throughout the entire business, data are collected from various source systems, such as enterprise resource planning systems and manufacturing execution systems, or the IoT. These data managed in the data lake are highly diverse, not only in structure (e.g., structured product data, semi-structured sensor data, and unstructured computer aided design (CAD) files), but also in their characteristics. These characteristics range from highly sensitive master data to voluminous IoT data of unknown quality.

2) Various time requirements exist in the data lake. In regular intervals, data are extracted from source systems and transferred to the data lake, where they are processed in periodic batches. Some of these data should be available within an hour, others need to be processed once a day or less. At the same time data from sensors arrives as data stream. These data should be processed immediately, to enable quick and timely reactions to, e.g., malfunctions in the manufacturing process. Also, they should be stored for later use as batch data. Thus, both batch and stream processing are of importance in this data lake.

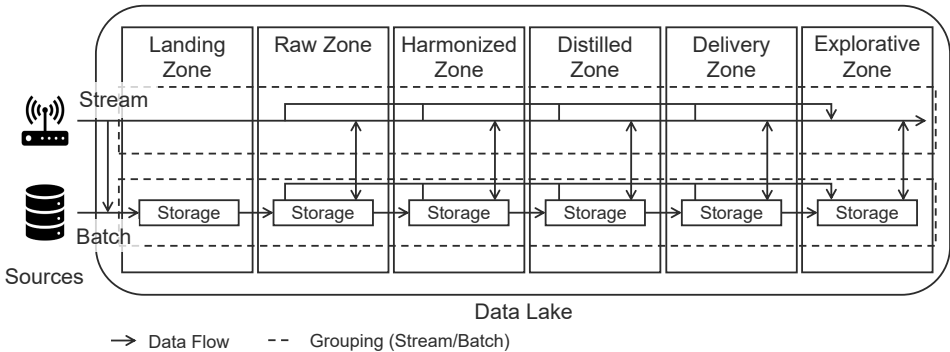


Fig. 6: An excerpt of the resulting data lake architecture, including data organization and data flow.

3) The data lake is the enterprise's central data repository that is accessed for a wide variety of use cases. These range from advanced analytics [Bo09], e.g., data mining and machine learning, to traditional reporting and online analytical processing (OLAP). For example, data in the data lake might be used to train a machine learning model to improve the quality of manufactured products, and to create a report for the supervisor of a specific plant.

Step 2: Design Data Flow. As specified in Step 1, the manufacturer relies on both batch and stream processing for the used data. Thus, the chosen data flow concept has to support both of these processing modes. For this exemplary data lake architecture, we decided to use the hybrid processing architecture BRAID [Gi18] as the data flow concept. In this architecture, data ingested as a stream are both forwarded to a persistent storage and to a stream processing engine. This behavior is similar to that of the Lambda Architecture as briefly described in Sect. 5.1. However, the BRAID architecture allows to use results from batch processing in stream processing. For example, a machine learning model can be trained on batch data and used to classify data from the data stream. In addition, results can be stored persistently and are available for later use. These characteristics of BRAID align with the manufacturer's requirements. Data ingested in batches are stored in the persistent storage and processed in batches. This data flow concept is depicted in Fig. 6.

Step 3: Design Data Organization. Because data are frequently accessed and used, a segmentation of the data lake into different portions is needed in this scenario. As data of different structure are often combined and the availability of raw data is crucial, we decided to use a data zone architecture. Due to the decisions made for the data flow aspect, this architecture needs to support both batch and stream processing. We chose the zone reference model [Gi20] as it provides fitting zones for the envisioned use cases for both batch and stream processing. It also contains concepts for both data quality and data security, e.g., the protected part, or varying access rights for different zones. Fig. 6 depicts the zone reference model and its interaction with batch data and streaming data.

Zone	Characteristics that influence Data Modeling	Data Modeling Technique
Landing Zone	Temporary	Raw Format
Raw Zone	Large amounts of data	Raw Format
Harmonized Zone	Standardized Modeling Technique	Data Vault (Raw Vault), Link-based Integration
Distilled Zone	Standardized Modeling Technique, Use Case Dependence	Data Vault (Business Vault), Link-based Integration
Delivery Zone	Prepared for specific tools	Modeling according to needs
Explorative Zone	Modeling done by data scientist	Modeling according to needs

Tab. 3: Overview of Data Modeling Decisions

Step 4: Design Data Storage. As the data to be managed is highly diverse, the data storage concept for this data lake comprises multiple different storage systems. That way, data can be managed where their characteristics and usage are supported best. For example, sensor data are stored in time series databases that support effective time-oriented queries (such as aggregations over time) while unstructured data are stored in a distributed file system.

Step 5: Design Infrastructure. As mentioned in step 4, multiple different storage systems and tools should be used. We chose various tools for storage and processing from the Hadoop ecosystem, e.g., HDFS and Apache Spark. Other systems, RDBMS and NoSQL databases alike, are added to this core to support more data characteristics. In addition, parts of the data lake are realized on a cloud-based structure to give third parties access to the stored data, such as suppliers or even end customers.

Step 6: Design Data Modeling. The usage of a data zone architecture for the data organization results in different modeling techniques in the zones. This is to support the required characteristics of the zones in the zone reference model. While data in the Landing Zone and Raw Zone are kept in their original format, Data Vault is used for structured data in the Harmonized Zone and the Distilled Zone. Data Vault allows flexible, use-case-independent, and scalable modeling of data in data lakes [Gi19b]. In addition, link-based integration [GSM14] is used to link structured and semi-structured data to unstructured data. The Harmonized Zone uses Raw Vault, while the Distilled Zone is modeled in Business Vault to include business logic. Finally, data in the Delivery Zone and the Explorative Zone are modeled according to specific needs.

Step 7: Design Metadata as Enabler. To handle all the stored data and to enable their usage, metadata management is needed. As metadata are also data, steps 1-6 have to be performed for them as well. Metadata may be structured or semi-structured and are ingested in the same way as the data it belongs to (e.g., as data stream for streaming data). The data flow concept for metadata thus is the same as for normal data. The data organization is unsegmented for metadata, as they span across the zones. For data storage, we decided to manage metadata in a graph database to support their highly connected structure (e.g., lineage metadata is

connected to data sources, operations, and resulting data). As infrastructure, we decided on Neo4J¹³. The metadata are modeled using HANDLE, which can represent, but is not limited to, lineage metadata, zone affiliations, and access information [Ei20]. This way, data security & privacy and data quality metadata can be stored, too.

Step 8: Design Data Processes. Data processes need to be specified in two sub-aspects, data lifecycle processes and data pipelining processes.

1) Data lifecycle processes in the scenario are defined in accordance with [DA17]. These processes manage data in all steps of the data lifecycle, ranging from creation over storage, use, and enhancement, to disposal. In all of these steps, metadata are captured and stored with the data, e.g., lineage metadata about data's creation, or metadata on who accessed data. Due to space reasons, we cannot discuss the aspect of lifecycle management in more detail. However, appropriate measures to comply with the data security & privacy and data quality concepts are taken, such as access control and change management.

2) Data pipelining processes are heavily intertwined with the data zone model used in data organization. Data are ingested and buffered in the Landing Zone before extract-transform-load (ETL) processes forward them to the Raw Zone. From there on, further ETL processes move the data into the other zones. These ETL processes apply transformations to the data to make them fit for the zone they are moving into. For example, data may be transformed according to data vault when moving from the Raw Zone to the Harmonized Zone. These processes are also responsible to realize the defined data security & privacy and data quality concepts. For example, personal data moving from the Landing Zone into the Raw Zone have to be anonymized. Similarly, data moving from the Raw Zone to the Harmonized Zone must follow certain quality guidelines.

Step 9: Design Metadata as a Feature. The final step is to specify concepts for metadata as a feature. In the scenario, we use three concepts that provide features in addition to those of metadata as enabler, namely a data catalog [Ch14] to allow access of data.

This completes the configuration of an exemplary data lake architecture using the DLAF. As can be seen in the description above, the usage of the DLAF and the associated methodology enabled a structured decision-making process. In the industry case, the DLAF provided guidance on what aspects to include and how to choose appropriate concepts for their implementation. It thus ensured that every aforementioned aspect is included in the data lake architecture and that their interdependencies are considered. For example, we could define data modeling with respect to the chosen zone model, or adjust the data processes to the chosen metadata management. The DLAF enabled interdisciplinary collaboration between domain experts, IT, and data scientists at the manufacturer's site by providing a common understanding of what a data lake architecture should comprise. Overall, the definition of this exemplary data lake architecture shows both the guidance DLAF provides as well as its applicability.

¹³ neo4j.com/

6 Conclusion and Future Work

While various concepts refer to themselves as data lake architectures, none of them covers all aspects necessary for a functional data lake. Thus, we developed the data lake architecture framework (DLAF) to support the definition of a scenario-specific data lake architecture. The DLAF consists of nine data lake aspects to be considered, their interdependencies, and a methodology to choose appropriate concepts for each aspect. The evaluation showed that the DLAF can be applied in two ways: 1) It can be used to identify missing aspects in existing data lake implementations and provide pointers towards re-design of the architecture. Our discussion of existing real-world data lake architectures showed that important aspects had been forgotten during the architecture's definition, such as metadata management. We showed that the DLAF supports not only the evaluation of existing data lake architectures to identify such shortcomings, but also their extension towards comprehensiveness. 2) The DLAF can be used to define a novel comprehensive data lake architecture. We used it in a real-world industry case. The DLAF enables a structured, step-by-step decision process, while providing decision support for choosing appropriate concepts. As interdependencies between aspects are considered by the DLAF methodology, the concepts of the resulting data lake architecture are well-matched to each other.

For future work, we plan to further apply and evaluate the developed data lake architecture in practice. Furthermore, the implications of the DLAF for an enterprise-wide usage across multiple data lakes should be investigated.

References

- [Ba14] Ballard, C. et al.: Information Governance Principles and Practices for a Big Data Landscape. IBM, 2014.
- [Bo09] Bose, R.: Advanced analytics: opportunities and challenges. *Industrial Management & Data Systems (IDMS)* 109/2, pp. 155–172, Mar. 2009.
- [Ch14] Chessell, M. et al.: Governing and Managing Big Data for Analytics and Decision Makers. IBM, 2014.
- [Ch15] Chessell, M. et al.: Designing and Operating a Data Reservoir. IBM, 2015.
- [CY15] Casado, R.; Younas, M.: Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience* 27/8, pp. 2078–2091, June 2015.
- [DA17] DAMA: DAMA-DMBOK: Data Management Body of Knowledge. Technics Publications, 2017.
- [De19] Dehghani, Z.: How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh, 2019, visited on: 05/27/2019.

-
- [Ei20] Eichler, R. et al.: HANDLE - A Generic Metadata Model for Data Lakes. In: Proceedings of the 22nd International Conference on Big Data Analytics and Knowledge Discovery (DaWaK2020). 2020.
 - [Ge17] Gessert, F. et al.: NoSQL database systems: a survey and decision guidance. Computer Science - Research and Development 32/3-4, pp. 353–365, July 2017.
 - [Gi18] Giebler, C. et al.: BRAID - A Hybrid Processing Architecture for Big Data. In: Proceedings of the 7th International Conference on Data Science, Technology and Applications (DATA 2018). SCITEPRESS - Science and Technology Publications, pp. 294–301, 2018.
 - [Gi19a] Giebler, C. et al.: Leveraging the Data Lake - Current State and Challenges. In: Proceedings of the 21st International Conference on Big Data Analytics and Knowledge Discovery (DaWaK 2019). 2019.
 - [Gi19b] Giebler, C. et al.: Modeling Data Lakes with Data Vault: Practical Experiences, Assessment, and Lessons Learned. In: Proceedings of the 38th Conference on Conceptual Modeling (ER 2019). 2019.
 - [Gi20] Giebler, C. et al.: A Zone Reference Model for Enterprise-Grade Data Lake Management. In: Proceedings of the 24th IEEE Enterprise Computing Conference (EDOC 2020). 2020.
 - [Go16] Gorelik, A.: The Enterprise Big Data Lake. O'Reilly Media, Inc., 2016.
 - [GSM14] Gröger, C.; Schwarz, H.; Mitschang, B.: The Deep Data Warehouse: Link-Based Integration and Enrichment of Warehouse Data and Unstructured Content. In: Proceedings of the 2014 IEEE 18th International Enterprise Distributed Object Computing Conference (EDOC 2014). IEEE, pp. 210–217, Sept. 2014.
 - [HGQ16] Hai, R.; Geisler, S.; Quix, C.: Constance: An Intelligent Data Lake System. In: Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16). Pp. 2097–2100, 2016.
 - [Ho17] Houle, P.: Data Lakes, Data Ponds, and Data Droplets, Online, 2017.
 - [In16] Inmon, B.: Data Lake Architecture - Designing the Data Lake and avoiding the Garbage Dump. Technics Publications, 2016.
 - [JQ17] Jarke, M.; Quix, C.: On Warehouses, Lakes, and Spaces: The Changing Role of Conceptual Modeling for Data Integration. In (Cabot, J. et al., eds.): Conceptual Modeling Perspectives. Springer International Publishing AG, chap. 16, pp. 231–245, 2017.
 - [Li12] Linstedt, D.: Super Charge Your Data Warehouse: Invaluable Data Modeling Rules to Implement Your Data Vault. 2012.
 - [Lo16] Lock, M.: Maximizing your Data Lake with a Cloud or Hybrid Approach, tech. rep., 2016.

- [Ma17a] Martínez-Prieto, M. A. et al.: Integrating Flight-related Information into a (Big) data lake. In: Proceedings of the 36th IEEE/AIAA Digital Avionics Systems Conference (DASC). IEEE, 2017.
- [Ma17b] Mathis, C.: Data Lakes. *Datenbank-Spektrum* 17/3, pp. 289–293, Nov. 2017.
- [ML16] Madera, C.; Laurent, A.: The Next Information Architecture Evolution: The Data Lake Wave. In: Proceedings of the 8th International Conference on Management of Digital EcoSystems (MEDES). ACM Press, New York, New York, USA, pp. 174–180, 2016.
- [MM18] Munshi, A. A.; Mohamed, Y. A.-R. I.: Data Lake Lambda Architecture for Smart Grids Big Data Analytics. *IEEE Access* 6/, pp. 40463–40471, 2018.
- [Mu13] Muschalle, A. et al.: Pricing Approaches for Data Markets. In: International Workshop on Business Intelligence for the Real-Time Enterprise (BIRTE 2012). Pp. 129–144, 2013.
- [MW15] Marz, N.; Warren, J.: *Big Data - Principles and best practices of scalable real-time data systems*. Manning Publications Co., 2015.
- [NRD18] Nogueira, I.; Romdhane, M.; Darmont, J.: Modeling Data Lake Metadata with a Data Vault. In: Proceedings of the 22nd International Database Engineering Applications Symposium (IDEAS 2018). 2018.
- [RZ19] Ravat, F.; Zhao, Y.: Data Lakes: Trends and Perspectives. In: Proceedings of the 30th International Conference on Database and Expert Systems Applications (DEXA 2019). Pp. 304–313, 2019.
- [SD20] Sawadogo, P.; Darmont, J.: On data lake architectures and metadata management. *Journal of Intelligent Information Systems*/, 2020.
- [Sh18] Sharma, B.: *Architecting Data Lakes - Data Management Architectures for Advanced Business Use Cases*. O'Reilly Media, Inc., 2018.
- [St20] Stach, C. et al.: AMNESIA: A Technical Solution towards GDPR-compliant Machine Learning. In: Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020). Pp. 21–32, 2020.
- [Te15] Terrizzano, I. et al.: Data Wrangling: The Challenging Journey from the Wild to the Lake. In: Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR'15). 2015.
- [Za87] Zachman, J. A.: A framework for information systems architecture. *IBM Systems Journal* 26/3, pp. 276–292, 1987.
- [Zi15] Zikopoulos, P. et al.: *Big Data Beyond the Hype*. McGraw-Hill Education, 2015, ISBN: 978-0-07-184466-6.

FactStack: Interoperable Data Management and Preservation for the Web and Industry 4.0

Lars Gleim,¹ Jan Pennekamp,² Liam Tirpitz,¹ Sascha Welten,¹ Florian Brillowski,³
Stefan Decker^{1,4}

Abstract:

Data exchange throughout the supply chain is essential for the agile and adaptive manufacturing processes of Industry 4.0. As companies employ numerous, frequently mutually incompatible data management and preservation approaches, interorganizational data sharing and reuse regularly requires human interaction and is thus associated with high overhead costs. An interoperable system, supporting the unified management, preservation, and exchange of data across organizational boundaries is missing to date. We propose FactStack, a unified approach to data management and preservation based upon a novel combination of existing Web-standards and tightly integrated with the HTTP protocol itself. Based on the FactDAG model, FactStack guides and supports the full data lifecycle in a FAIR and interoperable manner, independent of individual software solutions and backward-compatible with existing resource oriented architectures. We describe our reference implementation of the approach and evaluate its performance, showcasing scalability even to high-throughput applications. We analyze the system's applicability to industry using a representative real-world use case in aircraft manufacturing based on principal requirements identified in prior work. We conclude that FactStack fulfills all requirements and provides a promising solution for the on-demand integration of persistence and provenance into existing resource-oriented architectures, facilitating data management and preservation for the agile and interorganizational manufacturing processes of Industry 4.0. Through its open-source distribution, it is readily available for adoption by the community, paving the way for improved utility and usability of data management and preservation in digital manufacturing and supply chains.

Keywords: Web Technologies; Data Management; Memento; Persistence; PID; Industry 4.0

1 Introduction

While the management and preservation of manufacturing data regularly play a crucial role to fulfill legal and contractual accountability requirements, today's industrial data management is frequently considered an overhead factor instead of a valuable tool for data reuse, e.g., in the context of process optimization. While many aspects of data reuse have been studied in prior work [Gl20d; Ka17; LGD20], low-overhead data management solutions for industry are missing to date [Pe19b]. In the following, we introduce a representative use case scenario in the aerospace domain to motivate the remainder of the paper.

¹ Databases and Information Systems, RWTH Aachen University, Germany · gleim@dbis.rwth-aachen.de

² Communication and Distributed Systems, RWTH Aachen University, Germany

³ Institute of Textile Technology, RWTH Aachen University, Germany

⁴ Fraunhofer FIT, Sankt Augustin, Germany

Data Management and Preservation for Aircraft Manufacturing. The manufacturing of parts in the aerospace industry has strict certification requirements throughout the manufacturing process and supply chain, requiring detailed data about each process step to be collected, validated, and archived for years. For example, US-American regulations require the secure storage of *type design* case files, comprising drawings and specifications, information about dimensions, materials, and processes, for more than 100 years [Fe06]. Such strict requirements make sophisticated data management and preservation systems indispensable. At the same time, managing data in compliance with such regulations is traditionally associated with significant costs due to overheads incurred e.g., through manual data handling and inspection processes [Po17]. Considering a modern aircraft manufacturing supply chain, massive amounts of production data need to be managed and preserved [Pe19c], involving human paper-based signature mechanisms and leading to high associated overhead costs. In contrast, the efficient digital collection, management, exchange, and preservation of this data could lead to significant cost savings and productivity gains as part of Industry 4.0, not only in aviation but in many industries producing safety-critical components or otherwise facing strict certification and data retention requirements (e.g., textile, food processing, or plastics industry).

Use Case Scenario. Especially the manufacturing of structural elements in the aerospace industry relies on a large variety of technical textiles, such as light-weight, yet stiff carbon-fiber-reinforced plastics. We consider a common and simple aerospace scenario as illustrated in Fig. 1. Manufacturer **A** produces a light-weight carbon-fiber wing profile **R-001**, manufacturer **B** produces airscrew **PX9**, both collecting manufacturing process information along the process. Manufacturer **C** assembles an airplane **A1-001**, employing wing profile **R-001** sourced from **A** and airscrew **PX9**, sourced from **B**. **C** further conducts regular maintenance work on airplane **A1-001** throughout its lifetime, collecting corresponding maintenance data throughout its lifetime. Although material and workpiece identifiers within individual companies are usually standardized and production process data are often collected locally, individual resources are typically allocated to a specific cost center within the company's management and ERP system and cannot be easily linked to information in external systems, e.g., about which product, workpiece, or application may have been used during the manufacturing process. When, e.g., **C** buys **R-001** from **A**, existing manufacturing data, such as collected by **A** during the production process, is seldomly or only insufficiently passed on. Additionally, data collected during later stages of the product lifecycle, such as the maintenance data collected by **C**, is typically not passed back throughout the supply chain although it may serve as a valuable tool, e.g., in the context of wear and fatigue analysis of parts and products. Today, especially product quality data is mostly shared on paper and typically discarded after respective quality checks have been passed. Additionally, the quality of fiber-reinforced products is typically controlled only after post-processing is finished and changes are no longer possible [Me12]. A product is then either certified for the intended application during quality control or scrapped, while it may be perfectly reusable for other subsequent applications.

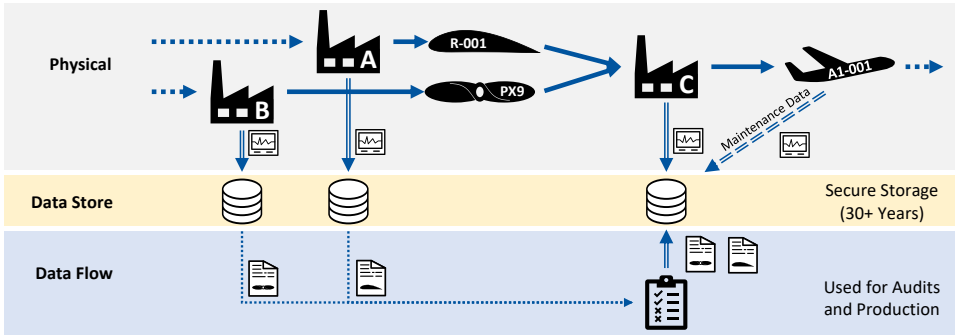


Fig. 1: Aerospace use case scenario: Manufacturer A produces wing profile R-001 and Manufacturer B airscrew PX9. Both collect quality data as part of their respective certification requirements during the process. Manufacturer C acquires the parts, employing them in the manufacturing of plane A1-001, keeps all related certification data and maintenance records in secure storage for 30+ years to comply with legal regulations.

Similar scenarios can be described for other domains in industry [Da19; Ni20; Pe19c; Pe20a; Pe20b]: Today, the collection, exchange, and preservation of data is frequently limited by the overhead cost of this data management (or fears of a loss of control over valuable data) while exactly the same data could be used for subsequent manufacturing process optimizations. Thus, an interoperable and principled data management and preservation system is needed to reduce data management overheads and therefore the associated costs and risks.

Principal Requirements. Data management and preservation for Industry 4.0 must enable the integration, exchange, and preservation of a wide variety of different types of data from all kinds of information systems employed throughout both the automation pyramid and the product lifecycle. Building upon the FAIR principles [Wi16] of scientific data management, an implementation should notably ensure that data is findable, accessible, interoperable, and reusable. Recent work has argued that these principles are equally applicable for data exchange throughout the supply chain and in Industry 4.0 [GI20b]. For the realization of these principles, a number of specific *services* that need to be provided by any data management solution have been identified in prior work [GD20a; GI20b; Hu00; Wi16], notably including:

1. **identification**, enabling globally unique and reliable referencing and citation of resources,
2. **versioning**, ensuring the immutability of individual resource revisions to avoid references from becoming incorrect due to content changes and enable change monitoring and state synchronization,
3. **persistence**, allowing individual resource revisions to be archived and persistently identified through so-called *persistent identifiers* (PIDs),

4. an **access** mechanism for resource retrieval and modification which should be open, free, and universally implementable,
5. **discovery** mechanisms, to make resources, metadata, and archives findable, and
6. accurate **metadata**, to ensure interoperability and reusability through clear semantics, e.g., by keeping track of data *provenance*, i.e., information about data origins, influences, and evolution over time.

To address these requirements, Gleim et al. [GI20b] recently proposed the FactDAG data interoperability model, for which we present a suitable implementation in this paper. Importantly, this implementation should further: provide the identified services in a manner that ensures **backward-compatibility** with existing resource oriented infrastructure and patterns as far as possible, be optionally adoptable, provide **interoperability** across software vendors and domains, employ non-proprietary, free, universally implementable and established **standards** whenever possible, and incur low overheads—both technical and otherwise—to support **sustainability**.

Contributions. To provide this implementation, we propose *FactStack*, an interoperable approach to data management and preservation based upon a novel combination of existing Web standards and tightly integrated with the HTTP protocol itself. Thereby, we directly realize the FactDAG data interoperability layer model, which we proposed in prior work [GI20a; GI20b], in a FAIR and interoperable manner, independent of individual software solutions and backward-compatible with existing resource oriented architectures. FactStack digitally supports data management throughout the full data lifecycle [Ba12; Co19] and directly integrates data management into the technology stack of the Web, instead of just using HTTP as an access mechanism. We further provide an open-source reference implementation of this approach, paving the way for its rapid adoption by the community and, subsequently, the proliferation of best practices for data management and preservation in digital manufacturing and supply chains. We demonstrate the scalability of the system to high-throughput applications and qualitatively highlight its applicability to industry using a real-world use case in the aerospace domain.

Paper Organization. The remainder of this paper is structured as follows. Sect. 2 provides an overview of related work and fundamental technologies. Sect. 3 conceptualizes our data management and preservation system, based upon open and standardized Web technologies. Sect. 4 then describes FactStack, our open-source implementation of this system, and evaluates its performance, before we discuss the impact of the proposed solution for data management and preservation in Sect. 5. Finally, we conclude our work in Sect. 6.

2 Related Work and Foundational Web Technologies

As related and foundational work, we first introduce existing data management solutions and the specific data characteristics and infrastructure requirements in the context of Industry 4.0. We then detail how Web technologies and standards provide fundamental primitives

and building blocks for the realization of interoperable data management. We discuss essential aspects of interorganizational interoperability and outline the role and importance of provenance information for reliable data reuse. Finally, we summarize the FactDAG data interoperability layer model [GI20b] as the theoretical foundation of our practical data management solution.

Data Management for Industry 4.0. The digital transformation already affects many areas of data management and preservation processes, ranging from the usage of collaborative file systems and well-known products, such as Dropbox or Google Drive, over collaborative model-based engineering environments [La19] to specialized version control systems (such as Git [AM19] or Mercurial [Ma06; RA12]). While industrial data management systems typically rely on database and data warehousing systems, the data in these systems is traditionally managed through external software and applications and not explicitly optimized for reuse. Therefore, data is typically only managed and accessed through respective application programming interfaces offered by the ERP (enterprise resource planning), MES (manufacturing execution system) or SCADA (supervisory control and data acquisition) systems, depending on the level of abstraction within the automation pyramid [In03]. Within organizations, data reuse is typically realized through use case specific ETL (extract, transform, load) processes, which require significant amounts of manual data cleaning and integration effort. Additionally, none of the individual data management systems are particularly suitable for the wide range of volume, velocity, variety, and veracity of heterogeneous data formats that need to be continuously managed, exchanged, and integrated at Internet scale for the full realization of Industry 4.0 [GI20b]. Pennekamp et al. [Pe19b] summarize, that an infrastructure for Industry 4.0 should be able to ingest, store, integrate, and query the heterogeneous production data (i.e., structured, semi-structured, or unstructured) in task-appropriate storage systems according to process-specific requirements and should also be generic and extensible for possible future needs. The authors further conclude, that existing systems are typically lacking semantic enrichment of data, e.g., using Semantic Web technologies [BHL01], which allows for them to be shared and reused across application, enterprise, and community boundaries, and enables the creation of machine-actionable knowledge. Based on their success in the realization of scalable, interoperable, and extensible enterprise solutions [BI13], Web technologies are already integral components of many existing data management systems (such as the aforementioned). In combination with Semantic Web technologies, they provide a promising basis for the development of interoperable and sustainable data management solutions for Industry 4.0 and the Web [GD20a].

Web Technologies for Interoperability. Interoperability in the Web is based on a number of fundamental standards, notably including: the global Domain Name System (DNS) [Mo87], the HTTP protocol and its implementation of the Representational State Transfer (REST) architectural pattern [Fi00], the Uniform Resource Identifier (URI), as well as its directly resolvable incarnation, the Uniform Resource Locator (URL) [BFM05]. Building on top of these foundations, Linked Data and the Semantic Web enable data interoperability on

the Web. Using the Resource Description Framework (RDF) [WLC14] data model and its serializations and enable machine-to-machine data interchange, the semantic enrichment of data, and the ability to interlink data across organizational boundaries. Deploying these standards supports interoperability. Notable standardized extensions of the basic HTTP protocol for distributed data exchange and management on the Web include (i) the Linked Data Platform (LDP) standard [SAM15], and (ii) the HTTP Memento protocol [VNS13]. The **Linked Data Platform** defines how Linked Data resources can be read and written using HTTP REST methods, i.e., HTTP GET, POST, PUT, PATCH, and DELETE. Besides resource *access*, the LDP enables the creation of containers that can be used to organize resources and to express relationships between them. Thus, it enables simplified resource *discovery*, as well as providing a mechanism to provide a dedicated metadata record for arbitrary Web resources using the HTTP `rel="describedby"` Link header. Using the LDP protocol, Linked Data and Web resources can be managed similarly to regular files in a local file system while enabling the augmentation of arbitrary resources with semantic *metadata*. A detailed introduction to the LDP can be found in [SAM15]. The **Memento protocol** introduces a mechanism to manage and retrieve persistent versions of Web resources by using timestamps as a resource version indicator and access key. Resource versions may be redundantly stored on multiple servers and managed independently of each other, enabling sustainable and distributed resource archiving [VNS13]. The Memento protocol provides primitives to address resource *versioning*, *persistence*, *access*, and *discovery*. As such, prior work already suggested the Memento protocol as a promising candidate for the implementation and standardization of data management and preservation systems [GD20a; Va14; Va18]. A detailed overview of the Memento protocol is provided in [VNS13].

Interorganizational Interoperability. An important factor limiting the adoption of interorganizational data exchange is uncertainty about the reliability of data, accountability, and liability questions for damages incurred by inaccurate data [Pe19a]. To this end, the concept of *data provenance* plays an important role in the realization of trust, accountability, and better interpretability of data and the processes that lead to their creation in collaborative manufacturing and supply chain systems. The term provenance, sometimes also called data lineage, refers to metadata regarding the formation history, origins, and influences that impacted the state of individual data. An open and extensible standard for provenance data is the W3C PROV data model (PROV-DM) for provenance interchange on the Web [MM13]. A primer on this model and its primitives can be found in [Gi13]. Provenance records are, e.g., successfully employed to build and analyze scientific workflows through process mining [Ze11], to ensure the reproducibility of such workflows [Ko10], to establish trust across heterogeneous sources of data [LLM10] and to further data reuse [Yu18]. Provenance data is special, in the sense that it is metadata that is relevant and collectible for practically any kind of resources and directly relates to the data authoring and management process. As such, it may serve as a generic kind of interoperable ‘glue’, relating resources throughout their formation history.

FactDAG Model. The conceptual *FactDAG* data interoperability model proposed by Gleim

et al. [GI20b] similarly employs data provenance to interlink resources and data throughout supply chain processes and in Industry 4.0. By using a persistent identification mechanism called *FactID*, FactDAG simultaneously addresses the requirements of *identification*, *versioning*, and *persistence*, constructing persistent identifiers from unique triples of global authority ID, internal resource ID and respective revision ID of a given immutable resource revision, also referred to as a *Fact*. The model further employs *Authorities*, entities (e.g., companies or organizations) that are responsible for Facts, *Processes*, which describe prototypical interactions with Facts, and *ProcessExecutions*, which refer to their instantiations and are introduced to capture individual influences and results (i.e., newly created Facts or Fact revisions). Additionally, a single relation (called *influence*, oriented forward in time) is used to express provenance relations between the elements of the FactDAG, thus constructing a provenance-based, directed acyclic graph of *Facts*, the *FactDAG*. Thus, the model allows tracing back the origins of Facts throughout time, revealing the resources, authorities, and processes involved in its conception and throughout the data management process. By globally and persistently identifying immutable revisions of resources, the model allows for information to be reliably referenced in global collaboration scenarios. The deep incorporation of provenance information into the model provides companies with a solid base of relevant metadata for the establishment of accountable, reliable, and sustainable data integration, even in interorganizational scenarios. For additional details, we refer to the specification of the FactDAG model [GI20b].

While we believe that the abstract FactDAG model provides a promising basis for the implementation of a data management and preservation system for Industry 4.0, it lacks both a concrete implementation, as well as a principled integration with best practices of data management to date. Thus, we propose a concrete implementation concept based on the fundamental data management lifecycle in the following section.

3 A Concept for Interoperable Data Management and Preservation

Data represent corporate assets with potential value beyond any immediate use, and therefore need to be accounted for and properly managed throughout their lifecycle [Fa14]. Various *data lifecycle models* [Ba12; Co19] have been proposed in recent years to serve as a high-level guideline for the data management process—from conception through preservation and sharing—to illustrate how data management activities relate to processes and workflows, to assist with understanding the expectations of proper data management and to ensure that data products will be well-described, preserved, accessible, and fit for reuse. The recurring elements of such models can be summarized in a five-step data management lifecycle model as illustrated in Fig. 2, consisting of the steps: (i) creation, processing, modification, and analysis, (ii) metadata management and data preservation, (iii) release and publishing of data, including proper licensing, documentation, etc., (iv) discovery for reuse of available data, and (v) the retrieval, curation, and capture of this data for subsequent processing. In the following, we consistently refer to these steps using the names provided in Fig. 2.

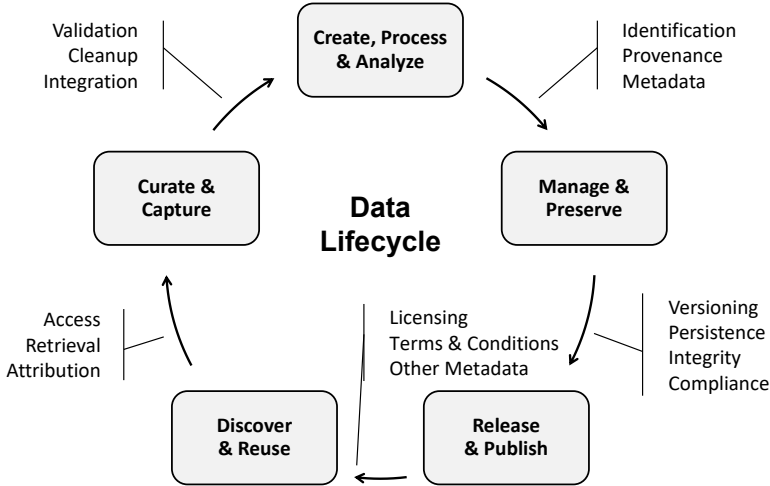


Fig. 2: The data management lifecycle consists of five steps providing an abstract model for data management processes, both in industry and academia. Adapted from [Ba12; Co19].

Traditionally, data management infrastructure is mainly employed as a kind of mediating service between the Release & Publish and the Discover & Reuse phase of the data lifecycle, while the remaining steps are carried out independently by human actors. In contrast, we aim to support the full data lifecycle process, integrating it directly with the fundamental infrastructure of the Web.

Technical Approach. Based on the principles of the FactDAG model [G120b], we strive to realize an interoperable data management and preservation system for usage throughout the data management lifecycle, the product lifecycle and the full supply chain, which satisfies the requirements identified in Sect. 1. As already motivated in Sect. 2, this system should build upon existing open Web standards whenever possible, ensuring compatibility and interoperability with existing systems and deployed solutions, as well as profiting from an ecosystem of developers with corresponding proficiency [St20] and the wide variety of available authentication and authorization mechanisms [TCS18]. The implementation should allow for incremental adoption, enabling the management and preservation of existing data according to the principles of the FactDAG model in an ad-hoc, on-demand fashion. Additionally, it should be backward-compatible, allowing clients that have no use for, do not support, or are unaware of data management principles in general, to simply ignore all related additional information. Provenance data should be collected and processed automatically whenever possible, especially during the Curate & Capture and Manage & Preserve phases, to minimize the amount of explicit markup and metadata management required and prevent easily avoidable user errors.

For the realization of these goals, we build upon two recent proposals by Gleim et al.: a PID

system employing dated URIs in conjunction with a resolution mechanism based on the HTTP Memento protocol [GD20b] (addressing aspects of data *identification*, *versioning*, *persistence*, and *access*), as well as an alignment of FactDAG provenance with the W3C PROV standard for provenance information [GI20c] (addressing aspects of data *provenance* and *discovery*). In the following, we outline and extend upon these proposals, integrating them with the W3C Linked Data Platform and other Web standards to form a comprehensive data management solution, addressing all requirements as formulated in Sect. 1.

3.1 FactID: Time-based Persistent Data Identification

To fulfill the *identification*, *versioning*, and *persistence* requirements as defined in Sect. 1 within the FactDAG model, a suitable persistent resource identification mechanism for the implementation of the FactID scheme is needed. As mentioned in Sect. 2, a FactID consists of the three components authority, internal resource ID and revision identifier. Inspired by the original FactID proposal [GI20b], we map all three components to a single URI to enable backward-compatibility with the Web infrastructure, as follows:

Authority *auth*. All data in the FactDAG model is placed under the exclusive and authoritative control of an organizational body, as identified by its global (but not persistent) authority ID $auth \in Authority$. We map *Authority* to the set of all DNS domain names [Mo87].

Internal ID *iID*. All resources available under the control of *auth* are identified by their respective internal resource ID $iID \in \mathcal{P}$. We map \mathcal{P} to the set of all URI Paths [BFM05] (including query and fragment suffixes). Combining *auth* and *iID* in a tuple creates a global (but not persistent) resource identifier, which we practically materialize as traditional HTTP URLs of the form `http://auth/iID`.

Revision ID τ . Individual resource revisions are further identified by their respective revision ID $\tau \in \mathcal{T}$. We map \mathcal{T} to the set of all RFC3339 [NK02] arbitrary precision UTC timestamps. While other revision identification mechanisms (such as content hashing) are conceivable, we specifically employ UTC timestamps due to their globally agreed-upon semantics. Timestamps are further unaffected by content-variations (e.g., due to content-negotiation) and allow for the intuitive ordering of resource revisions and their direct interpretation as time series data. Subsequently, the triple $(auth, iID, \tau) \in Authority \times \mathcal{P} \times \mathcal{T}$ yields a persistent global identifier – a FactID – for the immutable state (i.e., revision) of the resource identified by the tuple $(auth, iID)$ at the point in time τ . We employ the term *Fact* to refer to this immutable data state.

FactID URI Scheme. Many PID approaches require the assignment, registration, and management of PIDs outside of the Web infrastructure and already existing URL identifiers. This causes overhead for identifier mapping and discovery [Va14]. Thus, Gleim et al. [GD20b] proposed a system capable of reusing existing URLs as PIDs by combining dated URIs (for identification) with an HTTP Memento-inspired resolution mechanism (for versioning

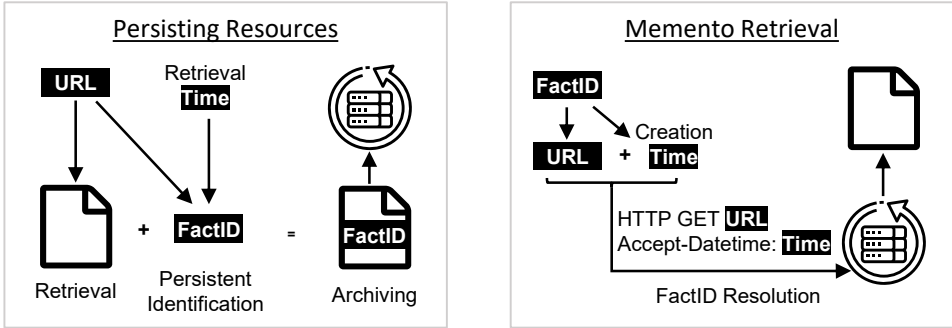


Fig. 3: Persisting and retrieving data using FactID. A FactID uniquely identifies a Fact (i.e., an immutable resource revision) by combining its URL with a timestamp. Such a FactID can be used to retrieve that Fact via the Memento protocol. Adapted from [GD20b].

and persistence). We employ this approach to realize a URI scheme for FactID through the following mapping:

While it is possible and common practice to resolve specific resource versions through URLs including HTTP query parameters, such an approach is hard to standardize in a backward-compatible manner. While a URL of e.g., the form `http://auth/iID/?v=τ` may be employed to uniformly express a persistent identifier according to the semantics of the FactID, the query parameter `v` is likely already used with different semantics in other contexts, creating the potential for naming conflicts. To avoid this problem, we adapt Larry Masinter’s ‘duri:’ dated URI proposal [Ma12] for the identification of specific resource revisions, resulting in the ‘factid:’ URI scheme: A FactID of the form `factid:τ:http://auth/iID` persistently identifies the immutable state of the resource `http://auth/iID` at the point in time `τ`, also referred to as a *Fact* or *Memento*. We refer the interested reader to Masinter’s RFC proposal [Ma12] for an additional discussion of the benefits and implications of employing dated URI.

HTTP-based Data Retrieval. To materialize and implement a practical resolution mechanism for such a FactID, we employ the HTTP Memento protocol as an *access* mechanism. Given a fixed ID, the resolution function $res : Authority \times \mathcal{P} \times \mathcal{T} \rightarrow Fact$ (with *Fact* as the set of all Facts) retrieves the Fact *f* identified by a given FactID through HTTP datetime negotiation via the HTTP Memento protocol [VNS13]. To maintain backward-compatibility, *res* defaults to resolving the current state of the resource identified by the tuple $(auth, iID)$, i.e., the URL `http://auth/iID`, if no revision ID is provided. The current resource state, as resolved via HTTP, may be lifted to a Fact by a consumer through incorporating the current timestamp as revision ID, as outlined in the original Fact construction procedure in [GI20b] and illustrated in the left half of Fig. 3. This way, it is possible to enable the on-demand incorporation of persistence into existing systems implementing REST semantics. Given such a `factid`, the original resource state may then be retrieved from an archive through an

HTTP GET request employing the Memento `Accept-Datetime` header with the Memento's creation time as specified in the τ part of the `factid` as depicted in the right half of Fig. 3. An overview of the different retrieval patterns and further features supported by the Memento protocol is given in [VNS13].

Data Persistence. As postulated by Kunze and Bermes [KB19], persistence (and analogously immutability) is purely a matter of service. It is neither inherent in an object nor conferred on it by a particular naming syntax but only achieved through a provider's successful stewardship of resources and their identifiers. Since the architecture of the HTTP Memento protocol "is fully distributed in the sense that resource versions may reside on multiple servers, and that any such server is likely only aware of the versions it holds" [VNS13], the service of data *persistence* may subsequently be provided by the authoritative data source, by any data consumer, by third parties such as governmental institutions or archiving providers or any number thereof, as long as volitional and legally permitted. By allowing for the *discovery* and retrieval of immutable data revisions over time, the Memento protocol further enables state synchronization between storage and archive locations as the data changes over time, thus additionally supporting redundant storage, e.g., for long-term data preservation. A detailed discussion of these applications may be found in [GD20b].

3.2 Automated Provenance Annotation and Distribution

As already discussed in Sect. 2, *provenance* is a particularly important category of *metadata* for data management. In alignment with the principal requirements identified in Sect. 1, we strive to minimize the metadata management overhead, by collecting *provenance* information automatically whenever possible. To ensure *interoperability* with existing tooling and reuse existing *standards*, we employ the recently proposed alignment [GI20c] of FactDAG provenance to the W3C PROV-O ontology standard [LSM13]. The mapping, illustrated in Fig. 4, thus allows for the expression of FactDAG provenance information as RDF metadata. For any given Fact f with FactID $fID = (auth, iID, \tau)$, the following provenance relation may be directly derived:

- f is an instance of the `prov:Entity` class.
- $auth$ is an instance of the `prov:Organization` class.
- f is `prov:wasAttributedTo` its authoritative source $auth$.
- If f is a direct revision of predecessor Fact f' , then f `prov:wasRevisionOf` f' .
- f is a `prov:specializationOf` its respective original resource, identified by the URL derived from the tuple $(auth, iID)$.

Additionally, information about any `prov:Entity` which was `prov:used` or `prov:wasGeneratedBy` a given `prov:Activity` may be automatically collected through the usage of a runtime library for *Fact* management, which we detail in Sect. 4. Nevertheless, further metadata and provenance information may have to be collected manually and can be added using RDF-compatible metadata vocabularies or other domain-specific ontologies, following the FAIR

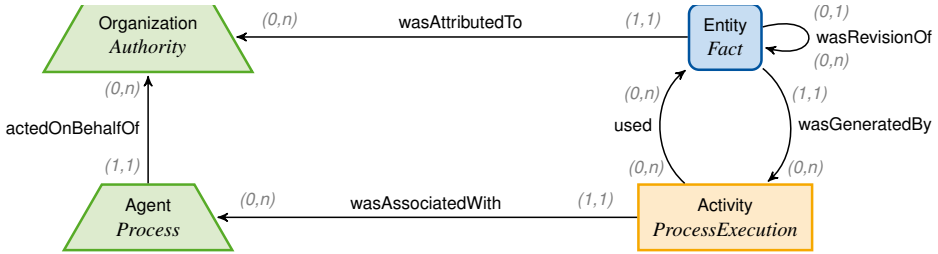


Fig. 4: The elements of the FactDAG model (in italic) and their provenance relations expressed using PROV-O primitives with corresponding (min, max) -cardinalities [Ab74]. The shapes represent the PROV core classes Entity, Activity, and Agent (with Organization as a subtype), respectively. Adapted from [Gl20c].

principle that metadata shall use a formal, accessible, shared, and broadly applicable language for knowledge representation and be described with a plurality of accurate and relevant attributes [Wi16].

3.3 Fact Discovery and Creation

The final missing conceptual component is a standardized discovery and read/write mechanism for data resource management. Due to its potential for interoperability with existing HTTP REST APIs and conceptual simplicity, we implement the Linked Data Platform specification [SAM15] for this task. Thus, resources (as identified by their respective HTTP URLs of the form `http://auth/iID`) can be organized in a hierarchy of LDP Containers within their authoritative source *auth*, enabling for resource discovery within it through exploration. In contrast, resource creation and modification are handled through the specified LDP HTTP REST methods.

To support a wide variety of structured, semi-structured, or unstructured data formats, including binary blobs of arbitrary file-type, the LDP specification further provides the option to augment Non-RDF LDP resources with a respective RDF metadata resource, linked through the HTTP `rel="describedby"` Link header, which we employ in practice, to store provenance information and further metadata. Both data and metadata can then be discovered through one single URL (or FactID respectively) and retrieved via HTTP.

Overall Concept. By combining dated URIs, the HTTP Memento protocol and the Linked Data Platform standard with PROV-O provenance and extensible RDF metadata, we ultimately propose a concept for semantic data management directly based on core technologies of the Web—URI, HTTP, and RDF—as illustrated in Fig. 5. By considering resource *versioning* and *persistence* as additional service layers of the basic Web technology stack and implementing them as extensions of URI and HTTP, we ensure *backward-compatibility* and *interoperability* with existing resources on the Web. By reusing existing

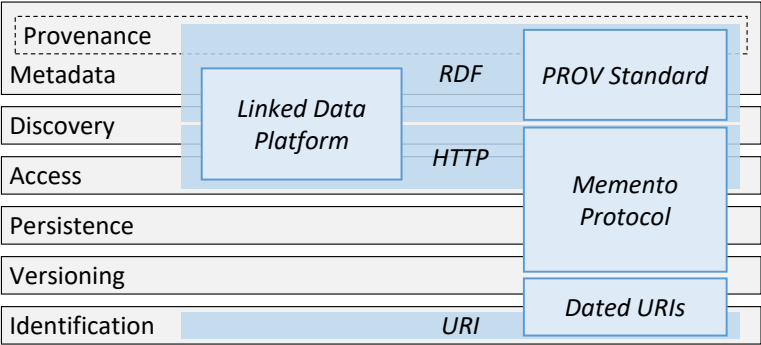


Fig. 5: The novel combination of existing protocols and Web standards provides a unified data management solution, addressing all principal requirements identified in Sect. 1.

standards where possible, and enabling on-demand resource versioning and persistence through consumers and third parties, the concept effectively addresses all requirements identified in Sect. 1, providing a promising foundation for its long-term *sustainability*. Based on this concept, we present our reference implementation of an interoperable data management and preservation solution for the Web and Industry 4.0.

4 FactStack: A Concrete Realization of the FactDAG Model

Following the conceptual approach presented in Sect. 3, we realize *FactStack* as a concrete open-source implementation of the FactDAG model for interoperable data management and preservation. Based upon the basic REST paradigm at the core of the Web, FactStack employs standardized and open Web technologies to provide a uniform data management API for arbitrary data resources on the Web, while enabling persistent data preservation through the HTTP Memento protocol.

Our realization consists of three open-source components, available for practical usage: A server component¹, adapted from the Trellis LDP project and implementing the LDP and Memento protocols for data storage and management, a JavaScript client library² simplifying both the interaction with the LDP server and the management of data provenance, as well as an optional broker, which enables real-time subscriptions to changes of LDP resources, i.e., newly created data revisions.

Data Storage. For the realization of a Fact authority, we employ a data storage server¹ based on the LDP implementation of the Trellis open-source project³. Trellis provides both direct integrations with a number of freely available storage backends, as well as the

¹ Available at: <https://git.rwth-aachen.de/i5/factdag/trellis>
² Available at: <https://git.rwth-aachen.de/i5/factdag/factlibjs>
³ <https://www.trellisldp.org/>

ability to integrate with existing information systems as its data store. The project further implements the HTTP Memento protocol [VNS13] for resource versioning, which we adapted to support RFC3339 [NK02] timestamps with up to nanosecond precision, as per the recent proposal of Gleim et al. [GD20b]. By default, all resources are identified by traditional URLs of the form `http://auth/iID`, their Mementos by corresponding FactIDs and revisions managed through the Memento protocol. To enable backward-compatible linking to Facts with standard URLs and resolution over plain HTTP, the server assigns an additional unique Memento URL *URL-M* (cf. [VNS13]) of the form `http://auth/iID/?v= τ` to each Fact. Finally, we implemented Memento headers to also be returned in response to LDP PUT and POST requests, as proposed by [GD20b], avoiding race conditions between competing resource updates and Memento header retrieval, thus ensuring efficient atomic resource updates.

Data Management. To guide the data management process in client applications, the *FactLib.js* library² mirrors the data lifecycle (cf. Fig. 2) in code. Facts are retrieved or created within the context of an activity and are subsequently registered as *used*, respectively *generated by* this activity, i.e., automatically recorded as corresponding provenance links. The library further handles the transparent and unified retrieval and creation of both RDF and Non-RDF Facts and their respective metadata, as well as automatically adding collected and inferred provenance information (cf. Sect. 3.2) as RDF metadata using the W3C PROV standard. For RDF resources, the provenance information is directly part of the resource stored in the LDP and can be found and retrieved by all clients that resolve the FactID to that resource. For binary resources, the Factlib.js library automatically discovers and manages metadata through the HTTP `rel="describedby"` Link header (cf. Sect. 3.3), retrieves it via HTTP and delivers it to the client application as part of the Fact. While authorities only store and provide access to Facts under their own control, clients can read and write from and to resources associated with different authorities. Clients may learn about Facts under the control of third-party authorities, e.g., by following provenance links (i.e., traversing the FactDAG), through explicit membership links provided by the LDP implementation or through other generic RDF triples or links. Each authority server may employ its own authentication and access authorization mechanisms, as well as providing its own data licensing terms, in order to maintain control over access to its data. Once a client successfully retrieved a resource, they may optionally (if legally allowed) archive it with any number of external Memento archiving providers (such as their own organization's) to serve as long-term persistence providers for arbitrary Facts. This distributed and usage-based archiving mechanism allows for flexible and use case driven trade-offs between persistence guarantees and associated costs, further contributing to the long-term *sustainability* of the data management approach as a whole. Retrieving Facts from third-party archives does, however, raise associated questions regarding authenticity and integrity, which we plan to consider in future work.

WebSocket Subscriptions. Since many applications in Industry 4.0 may profit from push-based real-time updates of changes to resources, e.g., to react to events with low latency,

a useful, practical feature consists of subscription support. Whenever new revisions of resources are created, a subscribed client receives a corresponding notification. Since for any pair of authority ID and internal ID, a series of data revisions could exist over time, all data within the FactDAG model is effectively *time-series data*. As such, every data point (as identified by authority and internal ID) is a stream of Facts and processing of facts is stream processing, which may, in turn, result in new Facts. To implement subscription support, we employ a broker-based approach to communicating change notifications in *Activity Streams 2.0* [SP17] format using a *STOMP*⁴ message encoding and a WebSockets [MF11] transport.

Performance. Finally, we conduct a performance evaluation of a single-node deployment of the server application on a workstation with Intel i7-8700K CPU, 64 GB of RAM and NVMe SSD. We configure Trellis to store Mementos in the file system and employ a local Apache ActiveMQ Artemis⁵ broker to support resource subscriptions via its Stomp over WebSockets implementation. We measure the average response time for Fact creation using HTTP PUT requests under different loads via Apache JMeter⁶, as well as the average time until a change notification is received back by a subscriber which consists of a basic collection script based on `stomp.py`⁷. To simulate random access to resources, 100 000 different resources are initially created containing five RDF triples each, as may be expected for small resources, such as single sensor values. The local JMeter client application issues PUT request to the server to update resources randomly chosen from this pool to apply the desired load and only starts the measurement of the average response times after an initial warm-up period. The `stomp.py` script subscribes to all resources with the ActiveMQ broker and records the timestamps of received notifications and the associated resource. The notification time is computed afterward, by comparing the timestamp of the request with the recorded timestamp of the collection script. To guarantee independent measurements, the whole system including the stored data is reset after each measurement.

The results plotted in Fig. 6 indicate a relatively stable average response time of around 10 ms for throughputs of up to 1000 insertions per second. For a throughput up to approximately 580 requests per second, the average response time is between 6 ms and 12 ms and the subscriber receives the notification in under 60 ms after the response confirming the resource modification is received by the sender. Around the 600 requests per second mark, the performance of the ActiveMQ broker deteriorates significantly, stabilizing at a notification latency of roughly 1 s for 680 requests per second and above. We attribute this behavior to the performance of the collector. If the collector cannot keep up with the processing of the incoming messages, the broker performance may be significantly reduced, as documented by the ActiveMQ project.⁸ Therefore, the maximum achieved load does not indicate the maximum capacity of the broker, but only the performance with regard to a single collector, which may be achieved by multiple collectors independently. In a practical

⁴ <https://stomp.github.io/stomp-specification-1.2.html>

⁵ <https://activemq.apache.org/components/artemis/download/release-notes-2.14.0>

⁶ <https://jmeter.apache.org/>

⁷ <https://github.com/jasonrbriggs/stomp.py>

⁸ <https://activemq.apache.org/components/artemis/documentation/latest/slow-consumers.html>

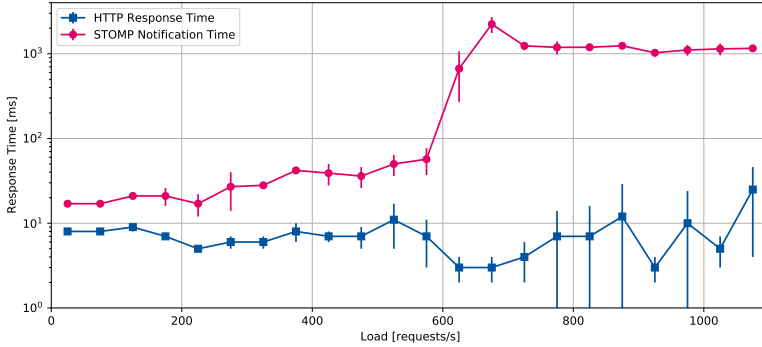


Fig. 6: Response times for random Fact creation under different loads as well as the time the system needs to notify a subscriber of the changed data (with 99 % CI). The results indicate sufficiently low and stable average response times for throughputs of up to 1000 insertions/s, as well as for notifications to the subscriber for throughputs up to 600 insertions/s.

scenario, a client would not typically subscribe to all resources, but only to the subset of those resources that are relevant to its immediate use case application. Thus, the ability to process up to 580 change notifications per second on a single client already provides a sufficient capacity for many practical scenarios. In order to expand the overall capacity beyond 1000 requests per second, e.g., for scenarios with multiple data producers and high update frequencies, users may instead profit from the horizontal scalability of Trellis LDP’s server architecture. In future work, we further plan to evaluate the performance of different data persistence backends and potential alternatives to Trellis for additional performance optimization.

With FactStack, we provide a concrete, open-source implementation of the FactDAG model for interoperable data management and preservation, facilitating the rapid adoption and evaluation by the community. After demonstrating the system’s scalability to high data-throughput scenarios, we discern its practical value for data management in Industry 4.0 in the following.

5 Applying the FactStack to Data Management in Industry 4.0

To ascertain FactStack’s value for practical application scenarios, we illustrate its data and control flow by mapping it to the research data management lifecycle presented in Fig. 2, resulting in the workflow shown in Fig. 7.

Starting with the Release & Publish phase, data is made available as resources on the Internet through regular HTTP Web services, each its own Authority identified by its domain name. In the Discover & Reuse phase, these resources may then be discovered either through existing

it may choose to either reuse the previously created Fact or to create an additional Memento analogously to the previous procedure.

During the Curate & Capture phase, an arbitrary number of resources may be collected, validated, cleansed, integrated, and subsequently provided as an input to the Create, Process & Analyze phase, in which data is created, modified, or deleted. To capture this process, it is modeled as a PROV Activity *A*, capturing all resources used as an input to or resulting from the execution of the activity as corresponding Entities. All Facts *S* provided as input to *A* are then related to it using the `prov:used` predicate, while any resources generated in the process are similarly persistently identified and immutably archived and related to *A* using the `prov:wasGeneratedBy` relation. Notably, if a generated resource *S* is a new revision of a previously existing resource *R*, this information is captured using the triple *S* `prov:wasRevisionOf` *R*.

During the Manage & Preserve phase, additional metadata may be added to the resource in order to capture more of its semantic context and provenance, while newly created resources are uniquely identified for future reference. Finally, resource and metadata (including any other applicable information such as licensing terms, etc.) are then stored together, identified by a single PID, during the Release & Publish phase, creating a new Memento or Fact in the process. Herein, the metadata may either be merged directly into the primary data – such as possible with RDF sources – or by adding it to the resource’s LDP Metadata resource accessible using HTTP content-negotiation or discoverable through HTTP `rel="describedby"` Link header (cf. Sect. 3.3), which enables structured RDF metadata to be stored for other text or binary file formats.

As all newly generated data are now persistently identified, immutably preserved, published, and discoverable on the Web, a full data management lifecycle was completed.

Use Case Application. Revisiting the use case example introduced in Sect. 1 and visualized in Fig. 1, we can now illustrate the concrete impact of data management and preservation using FactStack in Fig. 8. In this scenario, manufacturer **A**, identified by its authority domain name **A.com**, collects manufacturing data as part of the production of workpiece **R-001**, which it stores identified with internal resource ID `/R-001/qualitydata` in its data storage system Trellis A, creating a corresponding Memento at the point in time τ_1 . The internal Reporter process analyzes this data and creates a quality report with internal resource ID `/R-001/report` for this workpiece at the point in time τ_2 , certifying the part for usage in aerospace applications and again creating a corresponding immutable Memento. Company **C**, identified by its authority domain name **C.com**, now acquires workpiece **R-001** and retrieves the associated certification report Memento, recording its persistent FactID `factid: τ_2 :https://A.com/R-001/report`. **C** then stores a copy of this immutable Fact in its own data storage system Trellis C, where the Fact is still identifiable and retrievable through the standardized Memento protocol, using its original FactID, even if **A** deletes its copy or goes out of business. Even if a Fact becomes *globally* unavailable, knowing its FactID still allows for the derivation of basic metadata (cf. Sect. 3), as required by the FAIR data principles.

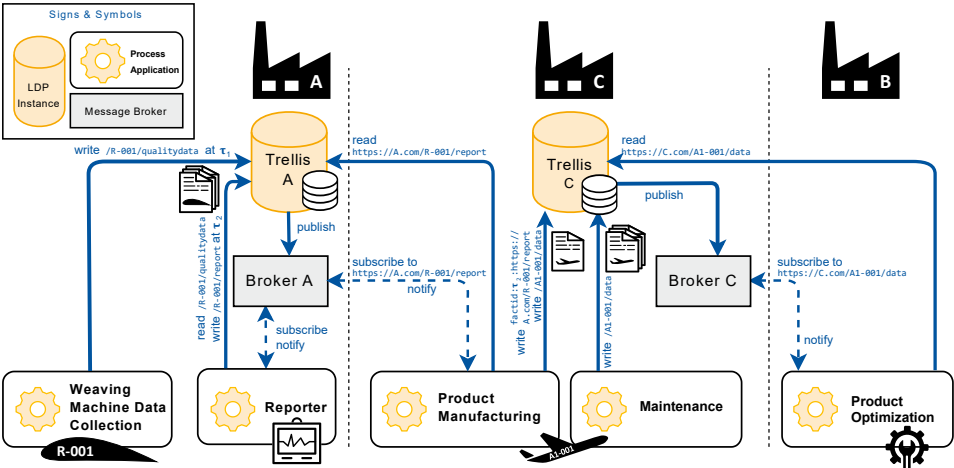


Fig. 8: To exemplify, FactStack enables continuous data sharing along the supply chain.

Additionally, **C** subscribes to the resource `https://A.com/R-001/report` to be automatically notified of any future updates to the resource, allowing for immediate reaction to change. **C** further aggregates and maintains all data related to airplane **A1-001** (both from Product Manufacturing and later Maintenance) in the RDF graph resource `https://C.com/A1-001/data`. Aircraft supplier **B** (which in this example does not provide any data using FactStack itself) then subscribes to this resource in order to continuously incorporate the maintenance data collected by **C** in its own Product Optimization process, thus (at least in theory) enabling the continuous improvement of its aircraft product designs.

Following FactStack’s data management lifecycle as illustrated in Fig. 7, metadata about each Process Execution in the described use case scenario is recorded through a corresponding PROV Activity. Links to all used (i.e., read) and generated (i.e., written) Facts (as identified by their corresponding FactIDs) are maintained as part of the RDF metadata of the corresponding generated resources. Subsequently, the origins and influences of any resource managed using FactStack can easily be traced back through the captured provenance relations, even across organizational boundaries and as resources change over time.

Discussion. To summarize, FactStack allows for the integration, exchange, and preservation of any type of data exchangeable on the Web and from any information system complying with the basic HTTP REST interface pattern. By implementing data identification, versioning, persistence, access, discovery, and metadata management through a novel combination of existing protocols and Web standards, it provides a backward-compatible and sustainable solution for data management and preservation. FactStack thus meets the system requirements posed in Sect. 1 and provides a promising solution for the management and preservation of the constantly evolving and diverse data of the Web and Industry 4.0.

Nevertheless, there are also some notable limitations. Although mandated by the FAIR data principles [Wi16], FactStack does not currently register nor index (meta)data in a searchable resource and does not enforce clear and accessible data licensing, nor domain-relevant community standards. Additionally, FactStack’s reliance on HTTP and its LDP and Memento protocol extensions can lead to high numbers of HTTP requests when managing data, since neither protocol supports request batching. Especially for resource discovery and RDF metadata management, significant performance improvements could likely be accomplished through the usage of the SPARQL query and update language [PPG13; SH13].

6 Conclusion and Future Work

In this work, we presented FactStack, an interoperable data management and preservation approach for evolving data on the Web and in Industry 4.0. Based upon open and tightly integrated with standardized Web technologies, FactStack realizes the FactDAG data interoperability model approach, providing on-demand support for persistent data archiving, identification, retrieval, and synchronization through an interoperable HTTP API, backward-compatible with existing REST services. By employing dated URIs according to the FactID scheme, we enable the persistent identification of arbitrary Web resources, resolved, managed, and preserved through a combination of the HTTP Memento and Linked Data Platform standards. We further implemented the semi-automated provenance collection with the W3C PROV-O ontology to enable the standard-compliant collection of data and process provenance as Linked Data.

To illustrate FactStack’s application in Industry 4.0, we focused on an exemplary, representative use case scenario in textile engineering for aerospace, highlighting corresponding opportunities for improved data management and preservation and interoperability. We support the practical adoption of the FactStack by releasing our implementation, which demonstrated scalability to high-throughput applications in the presented performance evaluation, as open-source software. FactStack promotes best practices for data management by directly supporting the full data management lifecycle and enables the continuous exchange and reuse of data using Web technologies throughout the supply chain and across domains, supporting the establishment of transparency and accountability through adequate and interoperable metadata and provenance management.

For future work, we plan to investigate the integration of the FactStack with existing enterprise resource planning and manufacturing execution systems to showcase FactStack’s universality and deployability. Additionally, future work should address related questions of authenticity, integrity, and trust within the FactDAG model, as well as improving the performance of the LDP server. Finally, we plan to implement and evaluate easy to use front-end applications for the intuitive collection of FactDAG data to simplify adoption for end-users and validate its merit in practical data management and preservation scenarios.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

References

- [Ab74] Abrial, J.-R.: Data Semantics. In: Proceeding of the IFIP Working Conference on Data Base Management. Elsevier, pp. 1–60, 1974, ISBN: 978-0-7204-2809-4.
- [AM19] Arndt, N.; Martin, M.: Decentralized Collaborative Knowledge Management Using Git. In: Proceedings of the 28th International Conference Companion on World Wide Web (WWW '19 Companion). IW3C2, pp. 952–953, 2019, ISBN: 978-1-4503-6675-5.
- [Ba12] Ball, A.: Review of Data Management Lifecycle Models, tech. rep., University of Bath, 2012.
- [BFM05] Berners-Lee, T.; Fielding, R. T.; Masinter, L. M.: Uniform Resource Identifier (URI): Generic Syntax, IETF RFC 3986, 2005.
- [BHL01] Berners-Lee, T.; Hendler, J.; Lassila, O.: The Semantic Web. *Scientific American* 284/5, pp. 34–43, 2001.
- [BI13] Bloomberg, J.: The Agile Architecture Revolution: How Cloud Computing, REST-Based SOA, and Mobile Computing Are Changing Enterprise IT. Wiley, 2013, ISBN: 978-1-118-41787-4.
- [Co19] Corti, L.; Van den Eynden, V.; Bishop, L.; Woollard, M.: Managing and Sharing Research Data: A Guide to Good Practice. SAGE, 2019, ISBN: 978-1-5264-8238-9.
- [Da19] Dahlmanns, M.; Dax, C.; Matzutt, R.; Pennekamp, J.; Hiller, J.; Wehrle, K.: Privacy-Preserving Remote Knowledge System. In: Proceedings of the 2019 IEEE 27th International Conference on Network Protocols (ICNP '19). IEEE, 2019, ISBN: 978-1-7281-2700-2.
- [Fa14] Faundeen, J. L.; Burley, T. E.; Carlino, J.; Govoni, D. L.; Henkel, H. S.; Holl, S.; Hutchison, V. B.; Martin, E.; Montgomery, E. T.; Ladino, C. C.; Tessler, S.; Zolly, L. S.: The United States Geological Survey Science Data Lifecycle Model, USGS Open-File Report 2013-1265, 2014.
- [Fe06] Federal Aviation Administration: Aircraft Certification Service Records, N1-237-05-003, 2006.
- [Fi00] Fielding, R. T.: Architectural Styles and the Design of Network-Based Software Architectures, PhD thesis, University of California, 2000.

- [GD20a] Gleim, L.; Decker, S.: Open Challenges for the Management and Preservation of Evolving Data on the Web. In: Proceedings of the 6th Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW '20). CEUR Workshop Proceedings, 2020.
- [GD20b] Gleim, L.; Decker, S.: Timestamped URLs as Persistent Identifiers. In: Proceedings of the 6th Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW '20). CEUR Workshop Proceedings, 2020.
- [Gi13] Gil, Y.; Miles, S.; Belhajjame, K.; Deus, H.; Garijo, D.; Klyne, G.; Missier, P.; Soiland-Reyes, S.; Zednik, S.: PROV Model Primer, W3C Working Group Note, 2013.
- [GI20a] Gleim, L.: FactStack: Interoperable Data Management and Preservation for the Web and Industry 4.0. In: RDA 16th Plenary Meeting — Poster Sessions. 2020.
- [GI20b] Gleim, L.; Pennekamp, J.; Liebenberg, M.; Buchsbaum, M.; Niemietz, P.; Knape, S.; Epple, A.; Storms, S.; Trauth, D.; Bergs, T.; Brecher, C.; Decker, S.; Lakemeyer, G.; Wehrle, K.: FactDAG: Formalizing Data Interoperability in an Internet of Production. IEEE Internet of Things Journal 7/4, pp. 3243–3253, 2020, issn: 2327-4662.
- [GI20c] Gleim, L.; Tirpitz, L.; Pennekamp, J.; Decker, S.: Expressing FactDAG Provenance with PROV-O. In: Proceedings of the 6th Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW '20). CEUR Workshop Proceedings, 2020.
- [GI20d] Gleim, L. C.; Karim, M. R.; Zimmermann, L.; Kohlbacher, O.; Stenzhorn, H.; Decker, S.; Beyan, O.: Enabling ad-hoc reuse of private data repositories through schema extraction. Journal of Biomedical Semantics 11/1, 2020, issn: 2041-1480.
- [Hu00] Hunter, G. S.: Preserving Digital Information: A How-to-do-it Manual. Neal-Schuman Publishers, 2000, isbn: 978-1-55570-353-0.
- [In03] International Electrotechnical Commission: Enterprise-control system integration - Part 1: Models and terminology, IEC 62264-1, 2003.
- [Ka17] Karim, R.; Heinrichs, M.; Gleim, L. C.; Cochez, M.; Porter, E.; Gioia, A. L.; Salahuddin, S.; O'Halloran, M.; Decker, S.; Beyan, O.: Towards a FAIR Sharing of Scientific Experiments: Improving Discoverability and Reusability of Dielectric Measurements of Biological Tissues. In: Proceedings of the 10th International Conference on Semantic Web Applications and Tools for Health Care and Life Sciences (SWAT4LS '17). Vol. 2042, CEUR Workshop Proceedings, 2017.
- [KB19] Kunze, J. A.; Bermès, E.: The ARK Identifier Scheme, IETF draft-kunze-ark-24, 2019.

-
- [Ko10] Koop, D.; Santos, E.; Bauer, B.; Troyer, M.; Freire, J.; Silva, C. T.: Bridging Workflow and Data Provenance Using Strong Links. In: Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM '10). Vol. 6187, Springer, pp. 397–415, 2010, ISBN: 978-3-642-13817-1.
- [La19] de Lange, P.; Nicolaescu, P.; Rosenstengel, M.; Klamma, R.: Collaborative Wireframing for Model-Driven Web Engineering. In: Proceedings of the 20th International Conference on Web Information Systems Engineering (WISE '19). Vol. 11881, Springer, pp. 373–388, 2019, ISBN: 978-3-030-34222-7.
- [LGD20] Lipp, J.; Gleim, L.; Decker, S.: Towards Reusability in the Semantic Web : Decoupling Naming, Validation, and Reasoning. In: Proceedings of the 11th Workshop on Ontology Design and Patterns (WOP '20). CEUR Workshop Proceedings, 2020.
- [LLM10] Li, X.; Lebo, T.; McGuinness, D. L.: Provenance-Based Strategies to Develop Trust in Semantic Web Applications. In: Proceedings of the 3rd International Provenance and Annotation Workshop on Provenance and Annotation of Data and Processes (IPAW '10). Vol. 6378, Springer, pp. 182–197, 2010, ISBN: 978-3-642-17818-4.
- [LSM13] Lebo, T.; Sahoo, S.; McGuinness, D.: PROV-O: The PROV Ontology, W3C Rec. 2013.
- [Ma06] Mackall, M.: Towards a Better SCM: Revlog and Mercurial. In: Proceedings of the 2006 Ottawa Linux Symposium. Pp. 83–90, 2006.
- [Ma12] Masinter, L. M.: The 'tdb' and 'duri' URI schemes, based on dated URIs, IETF draft-masinter-dated-uri-10, 2012.
- [Me12] Mersmann, C.: Industrialisierende Machine-Vision-Integration im Faserverbundleichtbau, PhD thesis, RWTH Aachen University, 2012, ISBN: 978-3-86359-062-8.
- [MF11] Melnikov, A.; Fette, I.: The WebSocket Protocol, IETF RFC 6455, 2011.
- [MM13] Missier, P.; Moreau, L.: PROV-DM: The PROV Data Model, W3C Rec. 2013.
- [Mo87] Mockapetris, P.: Domain names - concepts and facilities, IETF RFC 1034, 1987.
- [Ni20] Niemietz, P.; Pennekamp, J.; Kunze, I.; Trauth, D.; Wehrle, K.; Bergs, T.: Stamping Process Modelling in an Internet of Production. *Procedia Manufacturing* 49/, pp. 61–68, 2020, ISSN: 2351-9789.
- [NK02] Newman, C.; Klyne, G.: Date and Time on the Internet: Timestamps, RFC 3339, 2002.
- [Pe19a] Pennekamp, J.; Dahlmanns, M.; Gleim, L.; Decker, S.; Wehrle, K.: Security Considerations for Collaborations in an Industrial IoT-based Lab of Labs. In: Proceedings of the 3rd IEEE Global Conference on Internet of Things (GCIoT '19). IEEE, 2019, ISBN: 978-1-7281-4873-1.

- [Pe19b] Pennekamp, J.; Glebke, R.; Henze, M.; Meisen, T.; Quix, C.; Hai, R.; Gleim, L.; Niemietz, P.; Rudack, M.; Knape, S.; Epple, A.; Trauth, D.; Vroomen, U.; Bergs, T.; Brecher, C.; Bührig-Polaczek, A.; Jarke, M.; Wehrle, K.: Towards an Infrastructure Enabling the Internet of Production. In: Proceedings of the 2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS '19). IEEE, pp. 31–37, 2019, ISBN: 978-1-5386-8500-6.
- [Pe19c] Pennekamp, J.; Henze, M.; Schmidt, S.; Niemietz, P.; Fey, M.; Trauth, D.; Bergs, T.; Brecher, C.; Wehrle, K.: Dataflow Challenges in an *Internet* of Production: A Security & Privacy Perspective. In: Proceedings of the ACM Workshop on Cyber-Physical Systems Security & Privacy (CPS-SPC '19). ACM, pp. 27–38, 2019, ISBN: 978-1-4503-6831-5.
- [Pe20a] Pennekamp, J.; Bader, L.; Matzutt, R.; Niemietz, P.; Trauth, D.; Henze, M.; Bergs, T.; Wehrle, K.: Private Multi-Hop Accountability for Supply Chains. In: Proceedings of the 2020 IEEE International Conference on Communications Workshops (ICC Workshops '20). IEEE, 2020, ISBN: 978-1-7281-7440-2.
- [Pe20b] Pennekamp, J.; Buchholz, E.; Lockner, Y.; Dahlmanns, M.; Xi, T.; Fey, M.; Brecher, C.; Hopmann, C.; Wehrle, K.: Privacy-Preserving Production Process Parameter Exchange. In: Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC '20). ACM, pp. 510–525, 2020, ISBN: 978-1-4503-8858-0.
- [Po17] Ponemon Institute: The True Cost of Compliance with Data Protection Regulations, White Paper, Ponemon Institute, 2017.
- [PPG13] Passant, A.; Polleres, A.; Gearon, P.: SPARQL 1.1 Update, W3C Rec. 2013.
- [RA12] Rodriguez-Bustos, C.; Aponte, J.: How Distributed Version Control Systems impact open source software projects. In: Proceedings of the 2012 9th IEEE Working Conference on Mining Software Repositories (MSR '12). IEEE, pp. 36–39, 2012, ISBN: 978-1-4673-1761-0.
- [SAM15] Speicher, S.; Arwe, J.; Malhotra, A.: Linked Data Platform 1.0, W3C Rec. 2015.
- [SH13] Seaborne, A.; Harris, S.: SPARQL 1.1 Query Language, W3C Rec. 2013.
- [SP17] Snell, J.; Prodromou, E.: Activity Streams 2.0, W3C Rec. 2017.
- [St20] Stack Overflow: Developer Survey 2019, <https://insights.stackoverflow.com/survey/2019>, 2019 (accessed December 12, 2020).
- [TCS18] Trnka, M.; Cerny, T.; Stickney, N.: Survey of Authentication and Authorization for the Internet of Things. *Security and Communication Networks*/, 2018, ISSN: 1939-0114.
- [Va14] Van de Sompel, H.; Sanderson, R.; Shankar, H.; Klein, M.: Persistent Identifiers for Scholarly Assets and the Web: The Need for an Unambiguous Mapping. *International Journal of Digital Curation* 9/1, pp. 331–342, 2014, ISSN: 1746-8256.

- [Va18] Vander Sande, M.; Verborgh, R.; Hochstenbach, P.; Van de Sompel, H.: Toward sustainable publishing and querying of distributed Linked Data archives. *Journal of Documentation* 74/1, pp. 195–222, 2018, ISSN: 0022-0418.
- [VNS13] Van de Sompel, H.; Nelson, M.; Sanderson, R.: HTTP Framework for Time-Based Access to Resource States – Memento, IETF RFC 7089, 2013.
- [Wi16] Wilkinson, M. D.; Dumontier, M.; Aalbersberg, I. J. J.; Appleton, G.; Axton, M.; Baak, A.; Blomberg, N.; Boiten, J.-W.; da Silva Santos, L. B.; Bourne, P. E., et al.: The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data* 3/, 2016, ISSN: 2052-4463.
- [WLC14] Wood, D.; Lanthaler, M.; Cyganiak, R.: RDF 1.1 Concepts and Abstract Syntax, W3C Rec. 2014.
- [Yu18] Yuan, Z.; Ton That, D. H.; Kothari, S.; Fils, G.; Malik, T., et al.: Utilizing Provenance in Reusable Research Objects. In: *Informatics*. Vol. 5. 1, MDPI, 2018.
- [Ze11] Zeng, R.; He, X.; Li, J.; Liu, Z.; van der Aalst, W. M. P.: A Method to Build and Analyze Scientific Workflows from Provenance through Process Mining. In: *Proceedings of the 3rd Workshop on the Theory and Practice of Provenance (TaPP '11)*. USENIX Association, 2011.

Silentium! Run–Analyse–Eradicate the Noise out of the DB/OS Stack

Wolfgang Mauerer^{1,2}, Ralf Ramsauer³, Edson R. Lucas F.⁴, Daniel Lohmann⁵,
Stefanie Scherzinger⁶



Abstract: When multiple tenants compete for resources, database performance tends to suffer. Yet there are scenarios where guaranteed sub-millisecond latencies are crucial, such as in real-time data processing, IoT devices, or when operating in safety-critical environments. In this paper, we study how to make query latencies deterministic in the face of noise (whether caused by other tenants or unrelated operating system tasks). We perform controlled experiments with an in-memory database engine in a multi-tenant setting, where we successively eradicate noisy interference from within the system software stack, to the point where the engine runs close to *bare-metal* on the underlying hardware.

We show that we can achieve query latencies comparable to the database engine running as the sole tenant, but without noticeably impacting the workload of competing tenants. We discuss these results in the context of ongoing efforts to build custom operating systems for database workloads, and point out that for *certain* use cases, the margin for improvement is rather narrow. In fact, for scenarios like ours, existing operating systems might just be *good enough*, provided that they are expertly configured. We then critically discuss these findings in the light of a broader family of database systems (e.g. including disk-based), and how to extend the approach of this paper accordingly.

Keywords: Low-latency databases; tail latency; real-time databases; bounded-time query processing; DB-OS co-engineering

1 Introduction

The operating system is frequently considered boon and bane for the development of scalable service stacks. While general-purpose operating systems (like Linux) provide a great deal of hardware support, drivers and system abstractions, they have also been identified as a cause of jitter in network bandwidth, disk I/O, or CPU [Ar09; SDQ10; Xu13] when operating software services in cloud environments, where multiple tenants compete for resources. Naturally, this also affects the performance of cloud-hosted database engines [Ki15].

¹ Ostbayerische Technische Hochschule Regensburg, Germany wolfgang.mauerer@othr.de

² Siemens AG, Corporate Research and Technology, Munich, Germany

³ Ostbayerische Technische Hochschule Regensburg, Germany ralf.ramsauer@othr.de

⁴ Universität Passau, Germany edson.lucas@uni-passau.de

⁵ Leibnitz Universität Hannover, Germany lohmann@sra.uni-hannover.de

⁶ Universität Passau, Germany stefanie.scherzinger@uni-passau.de

Unacceptable noise and long-tailed latency distributions, but also the recent advances in hardware technology, have renewed interest in building database-specific operating systems. While historically, database and operating-systems research have been highly interwoven, the communities have parted ways in the past, and are just now rediscovering potential synergy effects (e.g. [Ca20; Mü20]). This has sparked immense interest in devising novel system architectures [KSL13], especially for database-centric operating systems kernels (e.g., [Ca20; Gi13; Gi19; MS19; Mü20]) that aim at deterministic performance. However, implementing an OS kernel is a herculean effort with tremendous follow-up costs, requiring substantial and largely duplicate effort for otherwise generic tasks, such as writing and maintaining device drivers, file systems, and infrastructure code, among others.

About This Paper. We take a fresh look at standard operating systems for low-latency/high determinism workloads, as they arise in real-time scenarios. Similar problems arise in cloud settings, where latency effects along the data path add up and can lead to substantial systemic problems, as Dean and Barroso have pointed out [DB13]. Rather than designing a new kernel from scratch⁷ to avoid noise and jitter, we follow an orthogonal approach, employing existing open-source components: Identify the root causes, analyse, and then address them as far as possible within the *existing* components. If necessary, enhance.

By vertical, cross-cutting engineering, we tailor the stack towards the needs of database engines, eradicate interference, and ultimately, reduce any noise-induced latencies in query evaluation. Our first results show that in many cases, a large degree of jitter is avoidable by the well-considered and purposeful employment of existing architectural measures – actually measures originally developed for other domains, such as embedded real-time systems. We present controlled experiments with an in-memory database engine running in a multi-tenant scenario on a number of different system software stack scenarios.

We focus on in-memory database engines as a specific (and deliberately narrow) use case, as they are often employed in domains for which deterministic latencies are essential [BL01], and thus considered a particularly convincing use case for developing specific operating-systems or even a bare-metal database stack [Bi20; Ca20; Gi13]. In this realm, our experimental setup, which is available as a Docker image for easy reproduction, can also serve as a baseline for researchers building special-purpose operating systems to compare their results against. In particular, we claim the following contributions:

- We perform controlled experiments with an in-memory database engine running on custom system software stacks based on existing open source components. By careful cross-cutting engineering, we modify this stack to eradicate interference, and to ultimately reduce any noise-induced latencies in query evaluation.
- We show that we can achieve the same performance using available operating systems as compared to running the database workload (near) *bare-metal*.

⁷ Whether to build a new operating system from scratch or whether to extend existing systems to cater to data processing needs has been an ongoing debate for decades [Gr78].

- We show that we can achieve the same performance in a multi-tenant scenario as compared to a database engine executing as the sole tenant without competing load.
- We voice doubts whether these specific scenarios can benefit from operating systems custom-designed towards database workloads, as they are currently being proposed.
- We discuss the potential generalisability of our approach to disk-based database engines, and systems involving I/O. In particular, we discuss opportunities that call for the joint efforts of the operating systems and database communities.

Structure. Our paper is structured as follows. We give an overview in Section 2, survey related work in Section 3, and present our experiments in Section 4. Their consequences are discussed in a more general context in Section 5. We conclude in Section 6.

2 Overview

We start with a brief summary of possible perturbations of an executing database workload by neighbourly noise, followed by an overview of the system software stack scenarios considered in this paper. In this section and beyond, by the term *kernel* we refer to the operating systems kernel (not the database kernel).

2.1 Sources of Noise

The three major sources of noise as observed by an unprivileged userspace workload (as compared to system services or the kernel) are (1) other processes and system services that compete for CPU usage, (2) CPU performance optimisations (caches, pipelines, . . .) that can usually not be disabled or controlled, and (3) contention of implicitly shared resources (memory bus etc.). The signature of such *systemic* noise is not necessarily distinguishable from the *intrinsic* noise of the application, that is, variations in run-time caused by data-dependent code paths, application-specific optimisations, and so forth.

Processes and system services. Multi-tasking operating systems manage M schedulable entities that compete for N processors, with $M \gg N$. Linux uses a *completely fair scheduling* (CFS) [Ma10] policy for regular processes, but also includes support for (soft) real-time scheduling via FIFO and round robin. The kernel can preempt most userland activities (depending on the preemption model statically configured at kernel build time), for instance upon the arrival of interrupts. It can also place *kernel threads* into the schedule that perform activities on behalf of the kernel (for instance, to support migrating processes across CPUs, to perform post-interrupt actions, etc.), and enjoy higher priority than regular processes, regardless whether these are governed by real-time policies. The interplay of these factors creates noise compared to an uninterrupted, continuous flow of execution of a single job.

CPU noise. Even given the uninterrupted execution of code on a CPU, pipelined and superscalar execution of code may lead to different temporal behaviour than would be

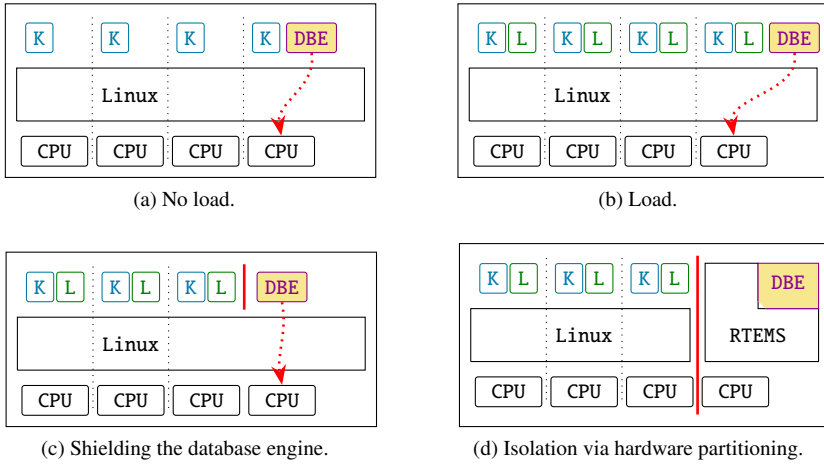


Fig. 1: System software stack scenarios.

achieved by a straightforward execution of assembly instructions, which manifests itself as another source of noise. Also, caching mechanisms (most importantly, the cache hierarchy that comes into play with memory references, but possibly also mechanisms like translation lookaside buffers used in virtual-to-physical address translation) cause (widely) varying latencies in accessing memory. This effectively adds noise.

Shared resources. Workloads executing on different CPUs are not entirely isolated from each other, but interact via shared resources (cache, memory, etc.) that are accessed via system buses. This even holds despite a possible logical partitioning of system components that we discuss later. While the overall situation (for instance, handling competing requests for bus usage) is deterministic from a system-global view, delays caused by competing requests manifest as noise when viewed from the perspective of an individual process.

2.2 Experimental System Configurations

The configurations of the system software stack, as used in our experiments, are visualised in Figure 1. For now, we treat the in-memory database engine (DBE) as a black box.

No Load. In the *no-load* scenario (Figure 1a), a single database engine executes on an otherwise quiet multicore system. The database payload is pinned to one CPU (c.f. the dashed arrow), to avoid perturbations, for instance caused by the scheduler moving the process across CPUs. However, standard system services, as limited to the bare necessity, and kernel threads required by the operating system proper (“K” in the figure) can execute on all CPUs, including the CPU dedicated to the database workload.

Load. In the *load* scenario (Figure 1b), additional tenants put the system under maximum strain. We simulate this payload (marked “L”) by running synthetic workloads on each CPU.

While the database workload is again pinned to one particular CPU, it is scheduled by the operating system alongside kernel threads and the described payload. In the load scenario, we assume the viewpoint of a cloud provider maximising the utilisation of the available resources, while serving all tenants equally. We therefore refrain from assigning the database workload higher priority than the payload generated by the competing tenants.

Load/FIFO. A variation of the *load* scenario uses standard Linux mechanisms to set a real-time scheduling policy for the database workload. All load processes fall under scheduling policy “other”, and compete for CPU resources as managed by the Linux standard scheduler. We place the database task in the real-time scheduling group `SCHED_FIFO`, so it can preempt any other userland tasks that execute on the CPU. However, the database task can still be preempted by the kernel, or by incoming interrupts.

Shielding. Another approach towards isolating the database workload from noise is to use CPU shielding (Figure 1c). This distributes all existing tasks and kernel threads on a given CPU to the rest of the system, and prevents utilisation of the shielded CPU by the standard scheduling for processes that are not explicitly assigned to this CPU.

We additionally make sure that incoming external interrupts only arrive at other CPUs. Nevertheless, main memory, buses, caches, etc. remain shared resources in the system, and accesses can induce additional noise that goes beyond the pure CPU noise. Additionally, the kernel can still preempt the single running userland task (for instance, when timers expire), and latencies can arise from administrative duties performed by the kernel on such occasions, or in the context of system calls issued by the task.

One set of measurements combines shielding and real-time priorities. This limits the kernel’s abilities to preempt the running userspace task. However, some caution needs to be administered: Not only the ability to preempt a running task, but also the amount of work performed in kernel context when a preemption occurs influences latency variance, and this amount is highly dependent on specific (static) kernel configuration settings.⁸ Isolation in this scenario is based on guarantees provided by the Linux kernel. This implies trust in a complex, monolithic code base, which is undesirable for safety-critical scenarios.

Partitioning. The strongest form of isolation that we consider in this paper (Figure 1d) relies on the Jailhouse hypervisor [Ra17]. Jailhouse can partition system hardware resources by establishing independent and strictly isolated computing domains. Jailhouse leverages extensions of the underlying system architecture which include essential virtualisation mechanisms for system partitioning, such as segregation of CPUs, memory and devices, as

⁸ Linux provides a tick-less mode, which eliminates periodic interventions by a regular timer (at frequencies ranging from 100Hz to 1,000Hz, depending on compile-time settings), but which may cause overhead on other occasions, because maintenance of data structures performed during such ticks must be performed “en block”.

well as additional extensions that allow to control the utilisation of shared resources, such as caches or system buses.

Jailhouse comes at a negligible performance overhead, as it does neither (para-)virtualise or emulate resources, nor schedule its partitions (*guests*) among CPUs. The virtual machine monitor only interferes in case of critical exceptions and access violations. This architecture can find application in multi-tenant database scenarios, described in [MKN12], and in particular, safety-critical scenarios, which require spatial-temporal isolation between tenants.

Bare-Metal Operation. Data center, cloud and high performance data processing systems often employ x86 server class CPUs, and we have argued before that such use cases benefit from bounded tail latencies. Other important use cases that require determinism are found in embedded systems, which are typically equipped with ARM CPUs. Consequently, our investigation addresses both, x86 and ARM.

Using a simplistic ARM core that is just capable enough for realistic database deployment reduces systemic noise that stems from multicore effects, as found on server-class x86 CPUs [PH90] to the bare minimum, (e.g. long pipelines, large caches, and strong interference on buses). This allows us to explore the *intrinsic* variations of our database workload.

We employ the in-memory database engine DBToaster [Ko14] (see also Section 4), a highly portable serverless database engine that requires a C++/STL run-time environment, but no other libraries or system services. Plain C++ can be executed without relying on an OS proper with moderate effort, but the STL requires (at least conceptual) support for threads and preemptive locking, as well as a full memory allocator. These requirements do not create a need to deploy it on top of a fully-fledged general-purpose operating system, such as Linux or Windows, but we deem the implementation efforts large enough to warrant a tiny operating system. Thus, we ported the database engine to RTEMS (real-time executive for multiprocessor systems) [BS14], a mature, tailorable embedded real-time operating system (with a 25-year development history) that finds deployment in systems ranging from IoT devices to Mars orbiters. Similar to unikernel approaches [Br15; Ma13], RTEMS and the database engine are linked together into one single executable. This binary can be either booted as stand-alone operating system on a bare-metal system, or (given low-level changes like the use of a custom bootloader and adaptations of the RTEMS kernel to Jailhouse) be executed in parallel to Linux on a partitioned system (as visualised in Figure 1d).

To reduce operating system noise as far as possible, we essentially limit RTEMS to providing only a console driver, and execute the database engine in a single thread, which eliminates the need for a scheduler. This configuration is supposed to reduce any OS noise to the bare minimum, and is *comparable to a bare-metal*⁹ main-loop style binary.

⁹ Bare-metal operation refers to code that runs without distinction between payload and OS close to the hardware, without intermediary layers. This in contrast to, for example, Ref. [RF18], which uses the term to denote code that runs without containers or virtual machines, but still relies on heavyweight, multi-million-LoC OS kernels.

3 Related Work

In this paper, we focus on in-memory database engines. We refer to [GS92] for an early overview of their architecture, and to [Fa17] for a more recent survey.

In real-time scenarios, where in-memory database engines traditionally play an important role [BL01], deterministic latencies are crucial. However, aspects such as consistency of query answers given transactional workload, or alternative tuple consumption strategies, are of no concern to the work presented in our paper, since we treat the database largely as a black box, and are interested in the overall system software stack.

What is indeed highly relevant for us is the existing work on worst-case execution time (WCET) of queries in in-memory databases [Bu05], which considers control-flow graphs through the code (in fact, in the presentation of our experiments, effects of different paths through control flow graphs during query processing actually become visible).

Also close to our work in both methodology and context is research on the influence of NUMA effects, focusing on in-memory database engines in particular. It is known that assigning threads to CPUs improves database performance, due to caching effects [Do18; Ki15; KSL13]. Similar studies of assigning database workloads to computational units can be found throughout database research, for instance in Refs. [DAM17; Po12]. Our experiments also assign threads to dedicated CPUs, and we benefit from data caching, but our motivation differs, as we *isolate* the database workload from harmful noise.

Databases operating in multi-tenant environments are another focus of our work. This differs from many benchmarks conducted in database research, where database workload often runs in isolation, while multi-tenant environments are closer to real-world conditions. Similarly, an overview over *performance isolation* for cloud databases is provided in [Ki15].

A systematic discussion of multi-tenant in-memory databases is provided in [MKN12]: From the viewpoint of a cloud provider, guaranteeing narrow service-level-agreements is a challenge, since the provider must cater to all tenants, while utilising the hardware resources. This mindset is also found in engineering for mixed-criticality systems [BD17; Ve07], where a critical workload (in our case, the database engine) must be shielded from noise (in our case, competing tenants), without cutting into the performance of the remaining workload.

In designing multi-tenant database engines, shielding tenants can be implemented on several levels in the system software stack. Aulbach et al. [Au08] enable multi-tenancy on the level of the database schema; by appropriately mapping between the tenants' schemas and the internal schema, tables may be transparently shared between tenants. By rewriting queries, the authors ascertain isolation between tenants in an otherwise standard database engine.

Narasayya et al. [Na13] also aim at resource isolation, for the database-a-service provider Microsoft SQL Azure. They explore virtual machine mechanisms in userland without relying on mechanisms provided by the kernel (and, consequently, not benefiting from the

guarantees provided by the OS kernel – for instance, some isolations are not possible in userland, such as access to shared buses and other resources).

Noll *et al.* [No18] discuss how to accelerate concurrent workloads inside a single database engine by partitioning caches. This feature is not targeted at multi-tenant databases *per se*, but applicable in general. However, this feature is specific to Intel CPUs. Further, it is not directly subject to control from userland, but exposed to applications by the `sysfs` pseudo-filesystem interface of the Linux kernel. Our x86-based RTEMS measurements in a Jailhouse cell actually use the same infrastructure to assign a portion of the cache to the system performing the measurements, which reduces variations in memory access times.

The general idea of using existing OS-level isolation mechanisms to reduce the amount of inference between latency (or otherwise) sensitive database workloads and the rest of a system has also been pursued by Rehmann *et al.* [RF18]: The authors use Docker containers to isolate database instances from system and competing payload noise. Their work essentially implements limiting the CPU quota available to tasks, and pinning database-relevant operations to specific CPUs in the system. Especially the latter is similar to some of our experiments, albeit we additionally include scheduling prioritisation and control the system noise on pinned CPUs with various measures. Thus, we make use of a richer toolset to achieve stronger levels of isolation, as our measurements show. In fact, containers are conceptually not intended to isolate a given workload from other workloads, but to provide a specific, probably restricted view of the system to a given workload.

Currently, there is renewed interest in building database-specific operating systems, partly motivated by such problems as unpredictability in performance. For instance, the MXKernel project [Mü20] proposes an alternative to the classic thread model, to cater to the demands of large-scale data processing. The DBOS initiative [Ca20] goes so far as to envision managing database-internal data structures inside the OS kernel. Further, there are suggestions to share the database cost model with the operating system [Gi13], to allow for more transparency and to ultimately arrive at better scheduling decisions.

Recent developments in modern hardware, and in particular modern memory technology, motivate database architects to re-evaluate the entire DBMS systems architecture and in-memory data structures [AP17; Re19; St07]. Over the years, research in this area has delivered promising propositions, e.g. [APD15; Ch18; GTS20; Le20]. In contrast, we evaluate how far *existing* technology will take us, given careful, cross-cutting engineering.

4 Experiments

We next describe the setup of our experiments, and then present our results. Our Docker image¹⁰, which we describe in Appendix 7, allows for inspection and reproduction.

Database Engine. We conduct our experiments with the in-memory database engine DBToaster [Ko14]. DBToaster can compile SQL queries to C++ code, which we then compile

¹⁰ Available online from <https://github.com/1fd/btw2021>.

(in a second step) for our target platform. The resulting executable is a single-threaded database engine that incrementally updates a SQL view given a tuple stream. DBToaster is thus a SQL-to-code compiler, designed to maintain materialised SQL views with low refresh latencies. Typical application scenarios would be in stream processing, such as algorithmic trading, network monitoring, or clickstream analysis¹¹. The DBToaster system and its theory have been prominently published (e.g. [KLT16; Ko10; Ko14; NDK16]).

We have created our own fork of the DBToaster code base (which is open source), with minor modifications for our experiments (e.g., buffering measurement data in memory, rather than writing directly to standard output). Our fork is part of our reproduction package.

Data and Queries. We consider two benchmark scenarios from the DBToaster experiments in [Ko13]. To be able to discuss the run-time results in greater detail, we focus on only a subset of queries. In particular, we exclude queries that display a high level of *intrinsic* variability in their latencies, where the computational effort between tuples can vary greatly, for instance because of nested correlated sub-queries and multi-joins. These queries are *per se* not well-suited for stream processing. The queries considered by us are listed in Figure 2.

Finance queries. The queries over financial data process a tuple stream with stock market activity; we chose three queries which use different relational operators: Query *cointone* (C1) is designed by us and serves as a minimal baseline. DBToaster can incrementally evaluate this query with constant-time overhead per tuple. The queries *axfinder* (AXF) and *pricespread* (PSP) each compute a join, a selection, aggregation, and in the case of *axfinder* also a group-by on the input stream. Here, we use the exact same query syntax as in [Ko13], as DBToaster has certain restrictions (e.g., no LEFT OUTER join). To be able to execute these queries on hardware devices with very limited memory, we use a base data set of 100 tuples¹², over which we iterate 5k times, yielding 500k data points. Since the query predicates do not filter on time-stamps, this does not affect query semantics.

TPC-H queries. We generated TPC-H data with the *dbgen* data generator, set to scale factor 4. We chose the queries Q6, Q1, and Q11a (shown in Figure 2) from the DBToaster experiments in [Ko13]. The queries perform selections, aggregations, and in the case of Q11a also a join.

Execution Platforms. For x86 reference measurements, we use a Dell PowerEdge T440. The T440 is equipped with a single 12 core Intel[®] Xeon[®] Gold 5118 CPUs and 32 GiB of main memory. For measurements on Linux, we use kernel version 5.4.38 (vanilla kernel as provided by kernel.org) as baseline, with the Preempt_RT real-time preemption patch.

Since delays are caused by parallel access to shared execution units and resources, symmetric multithreading (SMT) is a source of undesired high latencies and noise in real-time systems. Consequently, we deactivate SMT on our target, in accordance with the original DBToaster experiments. Furthermore, we deactivate Intel[®] Turbo Boost[®], as sporadic variations of

¹¹ See the project homepage at https://dbtoaster.github.io/home_about.html, last accessed January 2021.

¹² <https://github.com/dbtoaster/dbtoaster-experiments-data/blob/master/finance/tiny/finance.csv>

C1	<pre>SELECT count (1) FROM bids;</pre>	
axfinder	<pre>SELECT b.broker_id, SUM(a.volume+(-1*b.volume)) AS axfinder FROM bids b, asks a WHERE b.broker_id = a.broker_id AND ((a.price+((-1) * b.price)>1000) OR (b.price+((-1) * a.price)>1000)) GROUP BY b.broker_id;</pre>	pricesread <pre>SELECT SUM(a.price + (-1*b.price)) AS psp FROM bids b, asks a WHERE (b.volume > 0.0001 * (SELECT SUM(b1.volume) FROM bids b1)) AND (a.volume > 0.0001 * (SELECT SUM(a1.volume) FROM asks a1));</pre>
TPCH Q11a	<pre>SELECT ps.partkey, SUM(ps.supplycost * ps.availqty) AS query11a FROM partsupp ps, supplier s WHERE ps.suppkey = s.suppkey GROUP BY ps.partkey;</pre>	TPCH Q6 <pre>SELECT SUM(l.extendedprice*l.discount) AS revenue FROM lineitem l WHERE l.shipdate>=DATE('1994-01-01') AND l.shipdate<DATE('1995-01-01') AND (l.discount BETWEEN (0.06-0.01) AND (0.06+0.01)) AND l.quantity<24;</pre>
TPCH Q1	<pre>SELECT returnflag, linestatus, SUM(quantity) AS sum_qty, SUM(extendedprice) AS sum_base_price, SUM(extendedprice*(1-discount)) AS sum_disc_price, SUM(extendedprice*(1-discount)*(1+tax)) AS sum_charge, AVG(quantity) AS avg_qty, AVG(extendedprice) AS avg_price, AVG(discount) AS avg_disc, COUNT(*) AS count_order FROM lineitem WHERE shipdate<=DATE('1997-09-01') GROUP BY returnflag, linestatus;</pre>	

Fig. 2: SQL queries used in the experiments (queries from [Kol3], with the exception of C1).

the core frequency result in non-deterministic execution times for identical computational paths. We configure the CPUs in the highest possible P-State (performance setting) that guarantees a stable core frequency of 2.29 GHz.

For the shielding scenario, we try to remove all operating system noise from the target CPU. The Linux kernel provides multiple mechanisms for this purpose, of which we choose CPU namespaces that can be dynamically reconfigured during system operation.¹³

For the partitioned Jailhouse setup, we release one single CPU and 1 GiB of main memory from Linux, and assign them to a new computational domain. On that domain, we boot the RTEMS + DBToaster binary¹⁴ that runs in parallel to Linux. We use Intel’s Cache Allocation Technology (CAT), part of Intel’s Resource Director Technology, to partition last-level caches and exclusively assign 5 MiB of Level 3 Cache (L3\$) to the RTEMS + DBToaster domain. This mitigates noise (cache pollution) of neighboured CPUs, as the L3\$ is shared across all cores [In15].

For the ARM reference platform, we use a BeagleBone Black with a single-core Sitara AM3358, a 32 bit ARM Cortex-A8 processor and 512 MiB of main memory. In contrast to the powerful Intel server CPU, such ARM processors are typically found in embedded or industrial applications. We boot the RTEMS + DBToaster application directly on bare-metal.

¹³ Other mechanisms like CPU isolation at boot-time would provide a slightly higher level of isolation, but must be statically configured at boot-time, limiting the flexibility of the setup.

¹⁴ Getting DBToaster to run on RTEMS was not straightforward; along our trials, several fixes were proposed to open source systems, such as a decade-old bug revealed in GCC, as well as a bug identified in RTEMS.

Methodology. DBToaster logs a time-stamp for every N input tuples processed. This allows us to compute the *latency per N input tuples processed*, averaged over N tuples. While averaging is a sensible and established choice for throughput measurements to minimise overhead of the measurement intervention, we are interested in a precise characterisation of system noise vs. intrinsic variation of the core processing code, and therefore resort to measuring processing times on a *per-tuple* basis ($N = 1$).

We distinguish between two units of measurements: (1) time stamps obtained by the standard POSIX API (`clock_gettime` with `CLOCK_MONOTONIC`). This allows for nanosecond resolution, but also inflicts considerable overheads in the microsecond range, and introduces a noise level that is on par with the processing time proper for some of the simpler queries. Therefore, we extend DBToaster with the optional capability of (2) using x86 time stamp counter (TSC) ticks. While there are several problems and pitfalls associated with using the TSC on SMP configurations, and while the obtained measurement values cannot be converted to walltime without further ado [Ma10], TSCs are one of the highest-precision clock sources available on x86 hardware, and can be read from userspace without transition to kernelmode.¹⁵

As is a standard approach in settings like ours [BL01], we start measuring time once the input is in memory. In particular, we pre-load all tuples prior to stream processing, to exclude noise caused by I/O. Of course, in any real-world setting, the tuples would be read over peripheral communication channels, such as ethernet. To further avoid noise in our measurements, we have modified the code generated by DBToaster such that these time-stamps are cached in memory during query evaluation, in a pre-allocated array, rather than being continuously written to the standard output console.

Simulating tenant load. We simulate further tenants executing on the same system using the standard utility `stress-ng`, running 6 synthetic workloads.¹⁶ In Figure 1, we depict `stress-ng` running as additional load on the CPUs that are annotated with “L”.

4.1 Results

4.1.1 Noise and Determinism: Finance Queries

We begin our discussion of results for the finance queries. The time series in Figure 3 show observed latencies for processing each out of 500k input tuples. Red, labelled triangles mark the minima and maxima. Since almost all measured values fall into a comparatively

¹⁵ Using a high-resolution, low overhead time source is not necessary on our ARM reference platform because the time required to obtain a time stamp is negligible in comparison to the average processor performance, and our operating system has a flat memory and privilege model – that is, there is no distinction between kernel- and usermode on our near-bare-metal measurements on this platform.

¹⁶ (1) Binary search on a sorted array (exercises random memory access and processor caches), (2) matrix multiplication (to stress memory cache and floating point units), (3) compressing/decompressing random data (exercising CPU, cache, and memory), (4) randomly spread memory read and writes (to thrash the CPU cache), (5) sequential, random and memory mapped read/write operations (to exercise the I/O subsystem), and (6) timer interrupts at the frequency of 1 MHz (to induce continuous kernel/userspace transition due to interrupt handling).

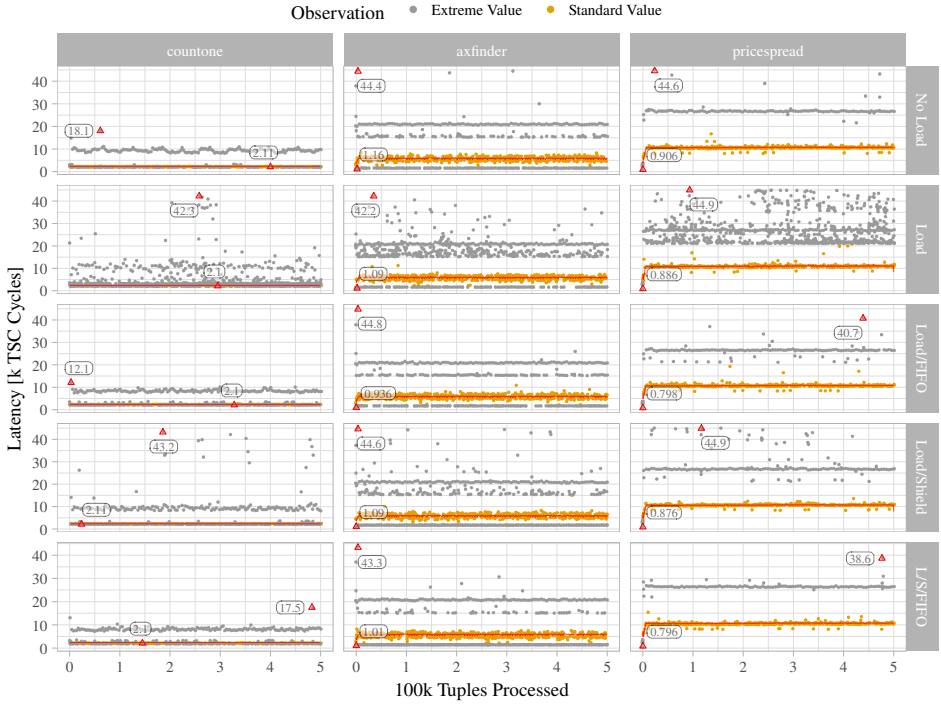


Fig. 3: Latency time series for finance queries on x86, using the high-speed time stamp counter (TSC).

narrow standard range, which would lead to massive over-plotting and loss of information, we colour all “extreme” measurement points that fall in the bottom 0.05% percentile, or that exceed the 99.95% percentile, in grey. We consider all other data points (marked ochre) as the normal observations. Note that such outliers have no noticeable influence when it comes to performance measurements, which usually concern query throughput, based on temporal averages, but are of paramount importance for real-time, bounded latency scenarios. For instance, the experiments in [Ko14] consider the query refresh times for processing batches of 1,000 tuples, and we compute a sliding mean window over 1,000 tuples as a consistency check; the resulting red line nicely reproduces the original DBToaster experiments [Ko14].

Each subplot of a given column corresponds to one system software stack scenario from Section 2. Almost all latencies are centred around the sliding mean value. However, a few outliers exceed the mean by a factor of about four.¹⁷

We have also tracked the average performance of the simulated other tenants, and found that it was essentially identical regardless of the measurement setup, which shows that improved

¹⁷ In the scenario discussed in this paper, the *maximum observed latency* is essential. Exceeding a threshold in industrial control scenarios might have severe consequences, from lost capital over destroyed machinery to bodily harm or loss of life, which can never be compensated by the fact the this does not happen *on average*.

determinism for a given workload does not necessarily decrease average throughput for non-real-time loads. Detailed data are available in the reproduction package.

While we consider queries of different intrinsic complexity, there is no direct relation between query complexity and noise – however, there *is* a relation between query complexity and *average* performance, as visible in the increasing latencies of the red line from left (simpler query) to right (more complicated query).

Query *countone* merely counts the number of input tuples processed so far (and is refreshed for each input tuple), whereas the other finance queries compute joins. As can be expected, the average latency for *countone* is distinctly lower. For the other queries, we can observe densely populated discrete “horizontal bands” that group the majority of all observed values. They correspond, based on an analysis of profiling data, to the main execution paths taken by the DBToaster-generated code (two main execution paths are a consequence of the “orderbook adapter” that distinguishes between the two types of input data, bids and asks). Also, when DBToaster-internal dynamic data structures grow in size (such as when buffering tuples for computing hash joins), additional DBToaster-intrinsic latencies incur.

The vertical spread of observations around these bands is an obvious visual noise measure. By comparing against the “Load” scenario, it is visually apparent that the different isolation mechanisms substantially reduce the observed jitter, typically to the level of an otherwise unloaded system. The strongest form of isolation, CPU shielding plus realtime scheduling (L/S/FIFO), produces latency distributions that are not only comparable, but even better in terms of maximum values than in the “No Load” scenario. The amount of noise decreases in order Load/Shield, Load/FIFO, and Load/Shield/FIFO. It might surprise that a shielded CPU performs worse than a CPU with additional load, but with a real-time prioritised task of interest. Recall that there is a complex interaction of kernel features as outlined in Sec. 2.2, and that, for instance, a larger set or possible preemption points, together with delayed kernel administrative work in a shielded scenario, may well compensate the advantages gained by exclusive CPU access.

While the measurements show a noticeable reduction of noise when using more advanced isolation techniques, the reduction of maximal latencies comprises only a factor of two.

4.1.2 Noise and Determinism: TPC-H Queries

The latency measurement results for the TPC-H queries are shown in Figure 4. While the general observations are identical – measured values concentrate around a few horizontal “bands”,¹⁸ and noise decreases with the various forms of systemic isolation – the behaviour of the queries under high load differs considerably from the “No load” and isolated case:

¹⁸ Notice that such bands may also be caused by system effects, and are then not necessarily present in all measurement combinations: For instance, the Load/Shield scenario in Figure 4 contains a band that disappears when FIFO scheduling is activated. Bands present in all scenarios are typically, but not necessarily, caused by the payload software. Such detail observations are not possible in measurements that average over observations.

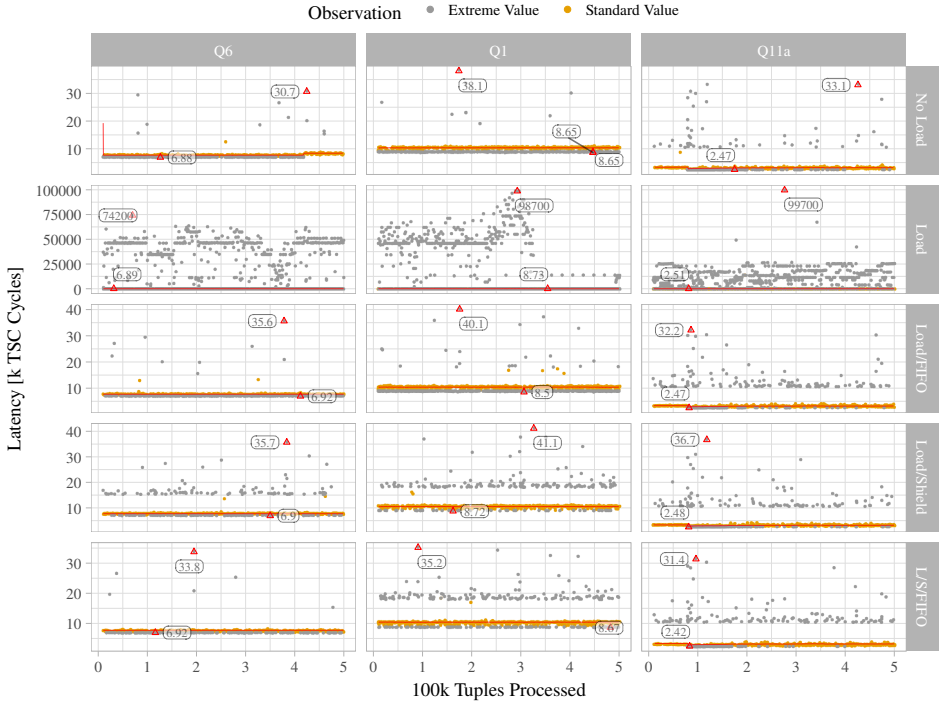


Fig. 4: Latency time series for TPC-H queries on x86, using the high-speed time stamp counter (TSC).

The difference in maximal latencies comprises more than three decimal orders of magnitude (observe the different scales of the vertical axes in the plots), and a similar statement can be made for the width of the spread around the running mean value, the latter again plotted with a red line. While such high variance has grave consequences for real-time systems, it is not even observable when throughput measurements are averaged.

So far, we have relied on visual means for characterising noise. For a quantitative measure, consider the set of observed latencies $\{\Delta t_i\}$ (each data point in Fig. 4 corresponds to one value of Δt_i). While we have focused on x86 so far, we will also consider ARM-based systems below. These platforms vary widely in their performance, and absolute values consequently require interpretation. It is therefore pertinent to consider *relative* deviations from the expected response time, which allows us to compare across platforms.

To this end, we define *spreads*, which are *not* influenced by the absolute processing speed. The *maximum spread* is given by $\max(\{\Delta t_i\})/\text{med}(\{\Delta t_i\})$, and *minimum spread* by $\text{med}(\{\Delta t_i\})/\min(\{\Delta t_i\})$, where $\text{med}(\cdot)$ denotes the median of the argument set. The quantities characterise the system-global relative span between a “typical” observed value, and the most extreme outliers in both directions. The results shown in the row labelled “TSC” of Fig. 5 quantitatively underlines this: Spread in the “Load” scenario typically exceeds

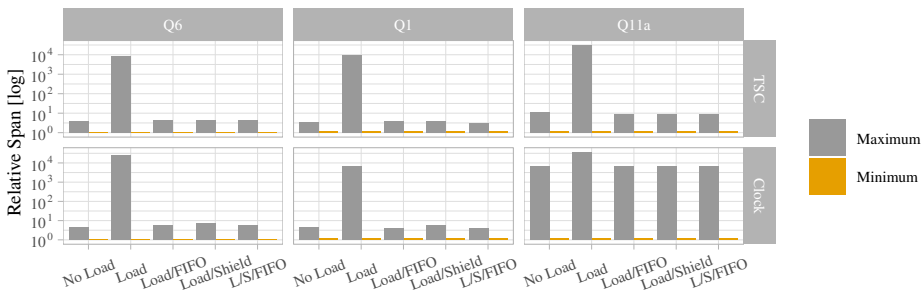


Fig. 5: Span of observations relative to median query latency of the TPC-H queries in Figure 4. Clock: Measured using `clock_gettime`. TSC: Measured using the high-resolution time stamp counter.

the spread in the isolation scenarios by orders of magnitude. The differences between the various isolation scenarios (and the “No Load” case) are much less pronounced, but can still encompass a factor of two or three (note the log-transformation of the y axis).

Recall that we distinguish between two units of measurements for latencies, (1) wallclock time in nanoseconds, and (2) x86 time-stamp counter ticks. The row labeled “Clock” of Figure 5 highlights another issue related to this fact that is mostly technical, but nonetheless requires careful consideration: how to perform the measurement itself. It shows the relative span for the *identical* measurement as considered in the other row, but this time using per-tuple latency measurements based on the POSIX API call `clock_gettime` offered as service by the Linux kernel (and often replaced by the lower-precision variant `gettimeofday`, in a good fraction of published performance measurements). Especially the maximum span can differ considerably among measurement variants. For TPC-H query 11a, it is even the major source of noise, as the right part of Fig. 5 shows.

Fig. 6 illustrates, for a subset of the isolation mechanisms, the increase in spread and noise distribution for clock-based measurements. It particularly highlights that even the mean throughput value (red line) is substantially influenced by the increased overhead.

4.2 The Role of CPU Noise

To a major extent, the previous experiments concern the control of noise introduced by the operating system and the presence of other tasks that compete for CPU time and other shared resources. Especially the scenario using CPU isolation, combined with a real-time scheduling policy, eliminates a substantial fraction of this noise. We now question how much of the remaining noise is caused by the executing CPU itself, and can thus be seen as an effective lower bound on any systemic noise.

We thus run our binaries as close to the bare-metal as possible, which reduces OS overhead. We perform these measurements on an ARM processor that we deem powerful enough to execute reasonable database operations, but that uses substantially fewer performance

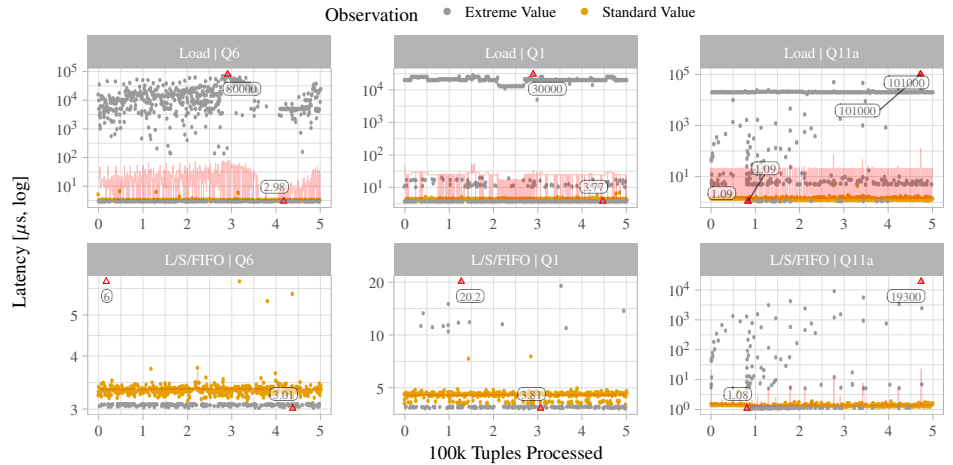


Fig. 6: Latency time series for TPC-H queries on x86, using a kernel-provided clock.

optimisations than x86 server-class CPUs (and, thus, suffers from less intrinsic noise). Our choice for an ARM CPU is not just driven by simplicity, though: Processors of this type are the most frequent choice in embedded systems and IoT devices, where low latency data processing is a common requirement (for instance, think of sensor-based systems that derive action decisions by combining previously measured values stored in a database with current data points). Our measurements are therefore representative for this large class of systems that we expect will gain even more importance in future applications. Of course, measurements on CPUs with drastically different capabilities cannot be directly compared, and this is not our desire: Instead, it is important to consider the *relative* difference between average and maximal latencies, and the span within measurements, as discussed below.

Fig. 7 shows latency time series for three finance queries. Again, observations centre around horizontal bands induced by the main execution paths, but the overall jitter is limited. The reduction compared to Jailhouse on x86 is quantified in Fig. 8.

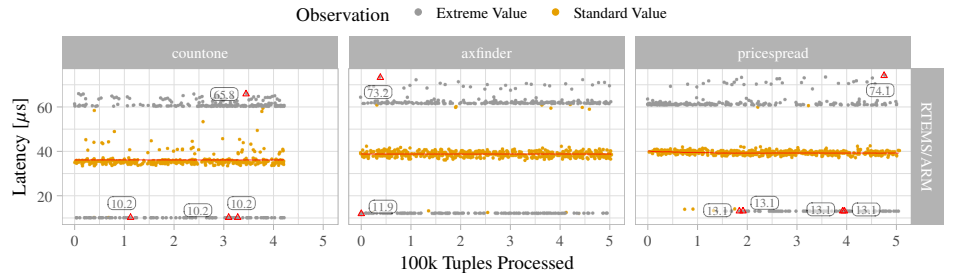


Fig. 7: Latency time series for finance queries on an ARM system (BeagleBone Black) using RTEMS. Red, labelled triangles represent minima and maxima (not necessarily unique).

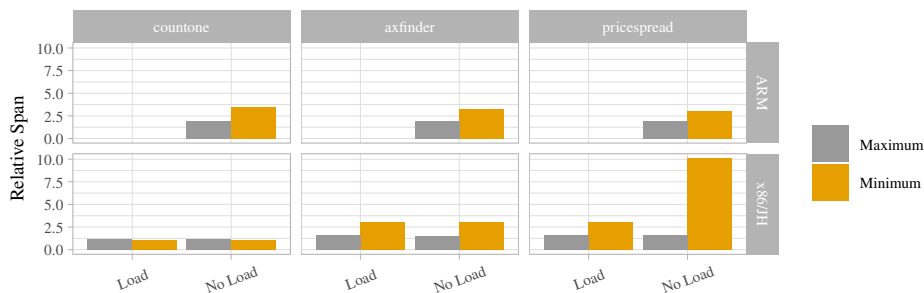


Fig. 8: Span of observations relative to median query latency of finance queries on bare-metal, on a high-end (x86) and low-end (ARM) CPU.

The summary for a second set of measurements shown in the bottom part of Fig. 8 represents bare-metal results obtained on the x86 CPU, but this time driven by an RTEMS kernel running inside a Jailhouse cell. Since the system is equipped with a total of 12 cores (compared to the single-core ARM), and only one of the cores is needed to run the database workload, we extend the measurement with an additional aspect that quantifies the aptitude of the setup to decouple latency-critical database operations performed by one tenant from other, perhaps throughput-oriented operations performed by other tenants. The spread is, as Fig. 8 shows, almost identical between the scenarios. This is also reflected in the time series shown in Fig. 9, which demonstrates that the results of the two configurations do not deviate in any substantial way. Since the isolation provided by Jailhouse does not only address execution timing, but also extends to other security and privacy related aspects of database workload processing, we deem this configuration a suitable basis for multi-tenant systems with strong separation guarantees.

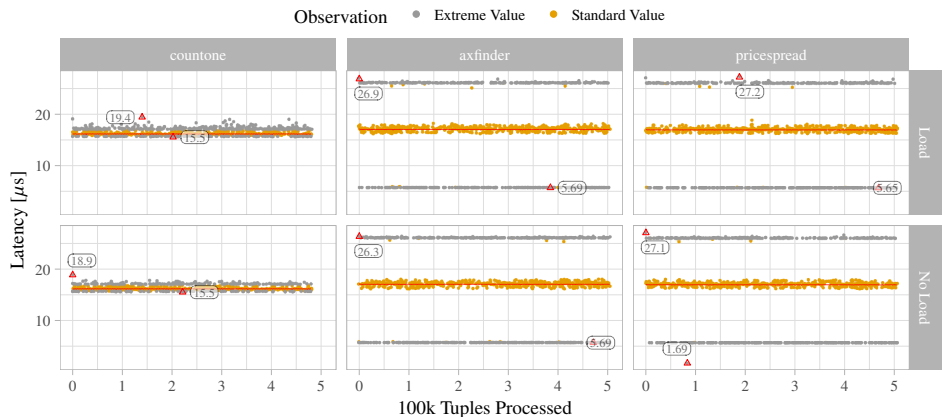


Fig. 9: Latency time series for finance queries on an RTEMS-based near bare-metal CPU provided by the Jailhouse hypervisor.

5 Consequences

Our experimental results show that – at least for our use-case of an in-memory database engine – building a database stack on a plain-vanilla Linux with custom settings can already compete with running close to bare-metal on the hardware. In our experiments we reached a state where the major source of noise turned out to be the interruptions to measure time (and noise) itself – any other sources for system-software induced jitter had been eradicated. In the following, we discuss some of these results in a broader context.

There’s life in the old dogs, yet. Our results suggest that it may not be necessary to design dedicated DB-aware operating systems *from scratch*. Rather, a prudent strategy to extend and enhance existing systems selectively may pay off equally well, and provide faster results. This is a recurring experience in the systems community: About a decade ago, the upcoming “multicore challenge” was supposed to render existing system-software designs unviable, and new kernel designs were deemed necessary [Ba09; Bo08; WA09]. It later turned out that existing system software could be scaled-up almost equally well by a systematic examination of their bottlenecks, which then could be fixed by employing standard techniques of parallel programming combined with *a few* novel abstractions [Bo10]. In a similar run–analyse–fix–approach, we have shown that existing system software, such as Linux, might just be *good enough* for many more database use cases, given proper configuration and adaptation. Of course, this does not invalidate the ongoing research on novel operating systems customised for database engines. Instead, the lesson to be learned here is that studying the *actual* reasons behind noise observed with existing operating systems is important. Only if we can pinpoint and understand the root causes, can we think of innovative solutions to these problems.

The cure might come by foreigners. Basically all of the measures we applied have originally been introduced into Linux to improve determinism and worst-case latencies not in database engines, but for the domain of embedded real-time systems: SCHED_FIFO, CPU shields, interrupt redirection, and PREEMPT_RT were developed and introduced to make Linux a suitable platform for mixed-criticality [BD17; Ve07] workloads in the real-time domain. This is also the main motivation behind partitioning hypervisors, such as Jailhouse. In our understanding, multi-tenant database engines that need to provide isolation and a guaranteed quality of service could (and should) be considered as (soft) real-time systems. Hence, operating-system solutions originally developed for the real-time domain might be a promising solution vector for the development of time-critical database systems. This underlines the necessity to rejoin the database and system communities.

Many challenges remain. Even though our results are promising, it should be pointed out that we eradicated only the CPU noise, and deliberately ignored I/O noise. Compared to the (narrow) case of a pure in-memory database, disk-based or otherwise I/O-intensive database stacks can be treated using the same measures as employed in this paper, but would face different challenges. In fact, most of Linux’s built-in real-time measures do not interact well with the I/O subsystem without further ado; it is often assumed that

the time-critical part can be decoupled from all I/O activities. While we have not yet examined disk-based database engines in this respect, we expect this to become a larger challenge that probably requires more invasive changes to the existing software stack. Direct I/O [Pe14] might be a promising way to approach this. Likewise, low-latency [Le19] or deterministic [SC03; Ya15] I/O scheduling have received a fair amount of attention outside the database community. External input via networks must consider additional stochastic parameters (e.g. unpredictable arrival times of data packets) that add to the complexity of the investigation. Real-time [KW05], or time-sensitive networking, and userland-based low-latency interaction with networking hardware can also be applied in database use-cases, albeit details must be left to future work.

What's next. Consider, as a specific and current example, how in-memory database engines can be extended with disk support – as, for instance, happens in the extension of Hyper to Umbra, where the authors propose to use SSDs for storage [NF20]. Especially parallel combinations of multiple SSDs promise RAM-like access performance.

However, parallel SSDs must be managed and driven. Database engines frequently aim at controlling block devices (at least to schedule access) from userland, since they have more complete usage pattern information than the OS. Yet this approach inevitably suffers from (at least) the amount of noise we have observed in our measurements, and advanced functionalities like RAID require substantial engineering effort. Operating systems provide such services as a commodity, but lack integration with the database query optimiser and its cost model. Additionally, operating systems are commonly optimised for throughput, so considerable tail latencies can be expected without adaptations. However, we are optimistic that moderate extensions of existing kernel mechanisms will combine the benefits of already existing infrastructure with little noise. This is important since increased determinism is beneficial to finding optimal query plans.

For all of the challenges listed above, we are optimistic that the required changes will be comparatively small compared to developing a new operating system from scratch.

6 Conclusion

We have shown that proper use of standard mechanisms of full-featured OSES can achieve database query latencies comparable to running an in-memory database engine directly on raw hardware. We reach a point where measuring time becomes the largest source of noise. By addressing challenges beyond CPU noise, we plan to bridge to the domain of real-time systems, and leverage techniques established for mixed-criticality systems which we apply to the database domain. After all, the underlying ideas match our scenario: One workload (the database engine) is to be shielded, without impairing the other workloads. We are confident that the respective research communities will enjoy many mutual benefits.

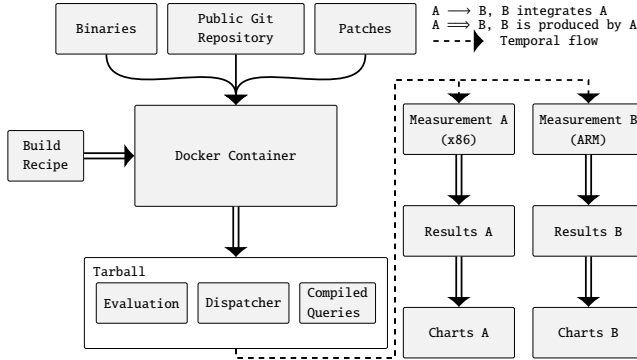


Fig. 10: Components and workflow of the reproduction package.

7 Appendix: Reproduction Package

Our publicly available reproduction package is based on Docker¹⁹. The process is illustrated in Figure 10: A Docker build recipe produces the docker container, and scripts that run therein produce a tarball with executables for all measurements in this paper. By transferring this tarball to a target, the experiments can be automatically executed, and charts generated.

We use binary sources for distribution-level software, and build other components (DBToaster, embedded compilers, RTEMS board support packages, . . .) from source using the latest released state, augmented with local patches, to address issues found during this work that relate to the RTEMS kernel, the Jailhouse hypervisor, the GNU C compiler, and DBToaster itself (we include patches as explicit diff files to make any deviations from upstream sources explicit without relying on git history inspection). Additionally, we do not rely on the long-term availability of external sources by providing a pre-built Docker image. It contains all sources and dependencies, and enables re-building the exact same binaries from source that we use for the measurements (of course, our peers may choose to build the Docker image from scratch, depending on the latest binaries).

Finally, we provide all raw measurement results for all system combinations considered in the paper, and all post-processing scripts to evaluate and visualise the data.

Acknowledgements. This work was supported by the iDev40 project and the German Research Council (DFG) under grant no. LO 1719/3-1. The information and results set out in this publication are those of the authors and do not necessarily reflect the opinion of the ECSEL Joint Undertaking. The iDev40 project has received funding from the ECSEL Joint Undertaking (JU) under grant no. 783163. The JU receives support from the European Union’s Horizon 2020 research and innovation programme. It is co-funded by the consortium members, grants from Austria, Germany, Belgium, Italy, Spain and Romania. We thank the DBToaster team, and Jan Kiszka for guidance on difficile technical issues related to Jailhouse on x86 systems.

¹⁹ Available online from <https://github.com/1fd/btw2021>.

References

- [AP17] Arulraj, J.; Pavlo, A.: How to Build a Non-Volatile Memory Database Management System. In: Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data. SIGMOD'17, pp. 1753–1758, 2017.
- [APD15] Arulraj, J.; Pavlo, A.; Dulloor, S. R.: Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD'15, pp. 707–722, 2015.
- [Ar09] Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing, tech. rep., University of California at Berkeley, 2009.
- [Au08] Aulbach, S.; Grust, T.; Jacobs, D.; Kemper, A.; Rittinger, J.: Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In (Wang, J. T.-L., ed.): Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGMOD'08, pp. 1195–1206, 2008.
- [Ba09] Baumann, A.; Barham, P.; Dagand, P.-E.; Harris, T.; Isaacs, R.; Peter, S.; Roscoe, T.; Schüpbach, A.; Singhanian, A.: The multikernel: A New OS Architecture For Scalable Multicore Systems. In: Proceedings of the 22nd ACM Symp. on Operating Systems Principles. SOSP'09, pp. 29–44, 2009.
- [BD17] Burns, A.; Davis, R. I.: A Survey of Research into Mixed Criticality Systems. ACM Comput. Surv. 50/6, 2017.
- [Bi20] Bittman, D.; Alvaro, P.; Mehra, P.; Long, D. D. E.; Miller, E. L.: Twizzler: a Data-Centric OS for Non-Volatile Memory. In: 2020 USENIX Annual Technical Conference. USENIX ATC'20, pp. 65–80, 2020.
- [BL01] Buchmann, A. P.; Liebig, C.: Distributed, Object-Oriented, Active, Real-Time DBMS: We Want It All - Do We Need Them (at) All? In: Annual Reviews in Control, pp. 147–155, 2001.
- [Bo08] Boyd-Wickizer, S.; Chen, H.; Chen, R.; Mao, Y.; Kaashoek, F.; Morris, R.; Pesterev, A.; Stein, L.; Wu, M.; Dai, Y.; Zhang, Y.; Zhang, Z.: Corey: An Operating System for Many Cores. In: 8th Symposium on Operating System Design and Implementation. OSDI'08, pp. 43–57, 2008.
- [Bo10] Boyd-Wickizer, S.; Clements, A. T.; Mao, Y.; Pesterev, A.; Kaashoek, M. F.; Morris, R.; Zeldovich, N.: An Analysis of Linux Scalability to Many Cores. In: 9th Symposium on Operating System Design and Implementation. OSDI'10, 2010.
- [Br15] Bratterud, A.; Walla, A.-A.; Haugerud, H.; Engelstad, P. E.; Begnum, K.: IncludeOS: A Minimal, Resource Efficient Unikernel For Cloud Services. In: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science. CloudCom'15, pp. 250–257, 2015.

- [BS14] Bloom, G.; Sherrill, J.: Scheduling and Thread Management with RTEMS. SIGBED Rev. 11/1, pp. 20–25, 2014.
- [Bu05] Buchmann, A. P.: Real-Time Databases. In (Rivero, L. C.; Doorn, J. H.; Ferragine, V. E., eds.): Encyclopedia of Database Technologies and Applications. Idea Group, pp. 524–529, 2005.
- [Ca20] Cafarella, M. J.; DeWitt, D. J.; Gadepally, V.; Kepner, J.; Kozyrakis, C.; Kraska, T.; Stonebraker, M.; Zaharia, M.: DBOS: A Proposal for a Data-Centric Operating System. CoRR abs/2007.11112/, 2020.
- [Ch18] Chandramouli, B.; Prasaad, G.; Kossmann, D.; Levandoski, J.; Hunter, J.; Barnett, M.: FASTER: A Concurrent Key-Value Store with In-Place Updates. In: Proceedings of the 2018 International Conference on Management of Data. SIGMOD’18, pp. 275–290, 2018.
- [DAM17] Dominico, S.; de Almeida, E. C.; Meira, J. A.: A PetriNet Mechanism for OLAP in NUMA. In: Proceedings of the 13th International Workshop on Data Management on New Hardware. DaMoN’17, pp. 1–4, 2017.
- [DB13] Dean, J.; Barroso, L. A.: The Tail at Scale. Commun. ACM 56/2, pp. 74–80, 2013.
- [Do18] Dominico, S.; de Almeida, E. C.; Meira, J. A.; Alves, M. A. Z.: An Elastic Multi-Core Allocation Mechanism for Database Systems. In: 2018 IEEE 34th International Conference on Data Engineering. ICDE’18, pp. 473–484, 2018.
- [Fa17] Faerber, F.; Kemper, A.; Larson, P.-Å.; Levandoski, J. J.; Neumann, T.; Pavlo, A.: Main Memory Database Systems. Found. Trends Databases 8/1-2, pp. 1–130, 2017.
- [Gi13] Giceva, J.; Salomie, T.-I.; Schüpbach, A.; Alonso, G.; Roscoe, T.: COD: Database / Operating System Co-Design. In: Sixth Biennial Conference on Innovative Data Systems Research, Online Proceedings. CIDR’13, 2013.
- [Gi19] Giceva, J.: Operating System Support for Data Management on Modern Hardware. IEEE Data Eng. Bull. 42/1, pp. 36–48, 2019.
- [Gr78] Gray, J. N.: Notes on Data Base Operating Systems. In: Operating Systems: An Advanced Course. Springer Berlin Heidelberg, pp. 393–481, 1978.
- [GS92] Garcia-Molina, H.; Salem, K.: Main Memory Database Systems: An Overview. IEEE Trans. on Knowl. and Data Eng. 4/6, pp. 509–516, 1992.
- [GTS20] Götze, P.; Tharanatha, A. K.; Sattler, K.-U.: Data Structure Primitives on Persistent Memory: An Evaluation. In: Proceedings of the 16th International Workshop on Data Management on New Hardware. DaMoN’20, 2020.
- [In15] Intel Corporation: Improving Real-Time Performance by Utilizing Cache Allocation Technology, <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf> (last accessed February 2021), 2015.

- [Ki15] Kiefer, T.; Schön, H.; Habich, D.; Lehner, W.: A Query, a Minute: Evaluating Performance Isolation in Cloud Databases. In: Performance Characterization and Benchmarking. Traditional to Big Data, pp. 173–187, 2015.
- [KLT16] Koch, C.; Lupei, D.; Tannen, V.: Incremental View Maintenance For Collection Programming. In: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. PODS’16, pp. 75–90, 2016.
- [Ko10] Koch, C.: Incremental Query Evaluation in a Ring of Databases. In: Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. PODS’10, pp. 87–98, 2010.
- [Ko13] Koch, C.; Ahmad, Y.; Kennedy, O. A.; Nikolic, M.; Nötzli, A.; Lupei, D.; Shaikhha, A.: DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views, tech. rep., EPFL, 2013.
- [Ko14] Koch, C.; Ahmad, Y.; Kennedy, O.; Nikolic, M.; Nötzli, A.; Lupei, D.; Shaikhha, A.: DBToaster: Higher-Order Delta Processing for Dynamic, Frequently Fresh Views. VLDB J. 23/2, pp. 253–278, 2014.
- [KSL13] Kiefer, T.; Schlegel, B.; Lehner, W.: Experimental Evaluation of NUMA Effects on Database Management Systems. In: Datenbanksysteme für Business, Technologie und Web (BTW) 2025. BTW’13, pp. 185–204, 2013.
- [KW05] Kiszka, J.; Wagner, B.: RTnet-A Flexible Hard Real-Time Networking Framework. In: 2005 IEEE Conference on Emerging Technologies and Factory Automation. Vol. 1, IEEE, pp. 456–464, 2005.
- [Le19] Lee, G.; Shin, S.; Song, W.; Ham, T. J.; Lee, J. W.; Jeong, J.: Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In: USENIX Annual Technical Conference. USENIX ATC’19, pp. 603–616, 2019.
- [Le20] Lersch, L.; Schreter, I.; Oukid, I.; Lehner, W.: Enabling Low Tail Latency on Multicore Key-Value Stores. Proc. VLDB Endow. 13/7, pp. 1091–1104, 2020.
- [Ma10] Mauerer, W.: Professional Linux Kernel Architecture. John Wiley & Sons, 2010.
- [Ma13] Madhavapeddy, A.; Mortier, R.; Rotsos, C.; Scott, D.; Singh, B.; Gazagnaire, T.; Smith, S.; Hand, S.; Crowcroft, J.: Unikernels: Library Operating Systems for the Cloud. SIGARCH Comput. Archit. News 41/1, pp. 461–472, 2013.
- [MKN12] Mühe, H.; Kemper, A.; Neumann, T.: The Mainframe Strikes Back: Elastic Multi-Tenancy Using Main Memory Database Systems On a Many-Core Server. In: Proceedings of the 15th International Conference on Extending Database Technology. EDBT’12, pp. 578–581, 2012.
- [MS19] Müller, M.; Spinczyk, O.: MxKernel: Rethinking Operating System Architecture for Many-Core Hardware. In: 9th Workshop on Systems for Multi-core and Heterogenous Architectures. EuroSys’19, 2019.

- [Mü20] Mühlig, J.; Müller, M.; Spincyk, O.; Teubner, J.: mxkernel: A Novel System Software Stack for Data Processing on Modern Hardware. *Datenbank-Spektrum* 20/3, pp. 223–230, 2020.
- [Na13] Narasayya, V.; Das, S.; Syamala, M.; Chandramouli, B.; Chaudhuri, S.: SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In: 6th Biennial Conf. on Innovative Data Systems Research. *CIDR'13*, 2013.
- [NDK16] Nikolic, M.; Dashti, M.; Koch, C.: How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In: *Proceedings of the 2016 International Conference on Management of Data. SIGMOD'16*, pp. 511–526, 2016.
- [NF20] Neumann, T.; Freitag, M. J.: Umbra: A Disk-Based System with In-Memory Performance. In: 10th Conference on Innovative Data Systems Research, Online Proceedings. *CIDR'20*, 2020.
- [No18] Noll, S.; Teubner, J.; May, N.; Böhm, A.: Accelerating Concurrent Workloads with CPU Cache Partitioning. In: *IEEE 34th International Conference on Data Engineering. ICDE'18*, pp. 437–448, 2018.
- [Pe14] Peter, S.; Li, J.; Zhang, I.; Ports, D. R. K.; Woos, D.; Krishnamurthy, A.; Anderson, T.; Roscoe, T.: Arrakis: The Operating System is the Control Plane. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. USENIX'14*, pp. 1–16, 2014.
- [PH90] Patterson, D. A.; Hennessy, J. L.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990, ISBN: 1558800698.
- [Po12] Porobic, D.; Pandis, I.; Branco, M.; Tözün, P.; Ailamaki, A.: OLTP on Hardware Islands. *Proc. VLDB Endow.* 5/11, pp. 1447–1458, 2012.
- [Ra17] Ramsauer, R.; Kiszka, J.; Lohmann, D.; Mauerer, W.: Look Mum, no VM Exits! (Almost). In: *Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPRT '17)*. 2017.
- [Re19] van Renen, A.; Vogel, L.; Leis, V.; Neumann, T.; Kemper, A.: Persistent Memory I/O Primitives. In: *Proceedings of the 15th International Workshop on Data Management on New Hardware. DaMoN'19*, 12:1–12:7, 2019.
- [RF18] Rehmann, K.-T.; Folkerts, E.: Performance of Containerized Database Management Systems. In: *Proceedings of the Workshop on Testing Database Systems. DBTest'18*, 2018.
- [SC03] Swaminathan, V.; Chakrabarty, K.: Energy-conscious, deterministic I/O device scheduling in hard real-time systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 22/7, pp. 847–858, 2003.
- [SDQ10] Schad, J.; Dittrich, J.; Quiané-Ruiz, J.-A.: Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB Endow.* 3/1, pp. 460–471, 2010.

- [St07] Stonebraker, M.; Madden, S.; Abadi, D. J.; Harizopoulos, S.; Hachem, N.; Helland, P.: The End of an Architectural Era: (It's Time for a Complete Rewrite). In: Proceedings of the 33rd International Conference on Very Large Data Bases. VLDB'07, pp. 1150–1160, 2007.
- [Ve07] Vestal, S.: Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance. In: Proceedings of the 28th IEEE International Real-Time Systems Symposium. RTSS'07, pp. 239–243, 2007.
- [WA09] Wentzlaff, D.; Agarwal, A.: Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. ACM SIGOPS Operating Systems Review 43/2, pp. 76–85, 2009.
- [Xu13] Xu, Y.; Musgrave, Z.; Noble, B.; Bailey, M.: Bobtail: Avoiding Long Tails in the Cloud. In: 10th USENIX Symposium on Networked Systems Design and Implementation. NSDI'13, pp. 329–341, 2013.
- [Ya15] Yang, S.; Harter, T.; Agrawal, N.; Kowsalya, S. S.; Krishnamurthy, A.; Al-Kiswani, S.; Kaushik, R. T.; Arpaci-Dusseau, A. C.; Arpaci-Dusseau, R. H.: Split-Level I/O Scheduling. In: Proceedings of the 25th Symposium on Operating Systems Principles. SOSP'15, pp. 474–489, 2015.

Data Management in Multi-Agent Simulation Systems

From Challenges to First Solutions

Daniel Glake,¹ Fabian Panse,¹ Norbert Ritter,¹ Thomas Clemen,² Ulfia Lenfers²

Abstract: Multi-agent simulations are an upcoming trend to deal with the urgent need to predict complex situations as they arise in many real-life areas, such as disaster or traffic management. Such simulations require large amounts of heterogeneous data ranging from spatio-temporal to standard object properties. This and the increasing demand for large scale and real-time simulations pose many challenges for data management. In this paper, we present the architecture of a typical agent-based simulation system, describe several data management challenges that arise in such a data ecosystem, and discuss their current solutions within our multi-agent simulation system MARS.

Keywords: Multi-agent simulations, Spatio-temporal data, Polyglot data management

1 Introduction

In the digital age, more and more data is available used to predict future conditions and effects emerging from potential (re)actions. A popular approach to make such predictions are simulation systems. They can be used to predict the course of catastrophic events, such as social-ecological changes [LWC18], nuclear disasters [Wa18] or epidemics [ZKC05] (e.g., to play through the effects of various measures), but can also be used to control, predict and evaluate everyday aspects, such as traffic with climate influence, topographic changes and individual-driven decision-making [WGC18]. One way to simulate such complex social-world processes is to use a multi-agent simulation (MAS) [WR15] in which the system models every individual by a separate agent interacting directly or indirectly with other agents or the considered world. Since MASs are temporal systems and often deal with spatial properties given by the represented world and locations of simulated objects, spatio-temporal data management is a crucial part of modern MAS systems such as GAMA [Gr13], NetLogo [WR15] or MARS (Multi-Agent Research and Simulation) [We19]. Due to the ongoing digitalization (e.g., through the Internet of things) and the growing availability of data (e.g., open data), simulations receive more and more attention while the heterogeneity and volume of useful data are continually growing. For short-term planning, such as city-wide traffic-jam forecasting [We19], simulation results must be

¹ Universität Hamburg, Vogt-Kölln-Straße 30, 22527 Hamburg, Germany {glake,panse,ritter}@informatik.uni-hamburg.de

² HAW Hamburg, Berliner Tor 7, 20099 Hamburg, Germany {thomas.clemen, ulfia.lenfers}@haw-hamburg.de

determined and aggregated quickly to provide value for decision support. In contrast, for long-term planning, e.g., infection spreading estimations [Ye06], the data management and simulation must be robust and offer sufficient capacity. Global sensitivity analyses further intensify these requirements. These circumstances cause several challenges in the data management of MAS systems, which we address in this paper from a general perspective before discussing some first solutions currently implemented in the MARS system.

This paper is structured as follows: In Section 2, we describe the typical components of a MAS system and how they are involved in the system's data management. Thereafter, we discuss open challenges for different data management aspects in Section 3 and describe in which way we address these challenges in MARS in Section 4. Finally, we discuss related work, conclude our paper and give an outlook on open research in Section 5.

2 Multi-Agent Simulation Systems

In this section, we describe the typical architecture of a MAS system (see Figure 1) and describe the individual components that interact within such a system. The architecture contains four main (represented by solid frames) and several optional (represented by dashed frames) components.

Simulation: The simulation component is the core of a simulation system. It receives a simulation model selected by the user and then loads all relevant input data from the data management component into the simulation's class model via the input adapter or the query mediator (see below). The simulation data can be categorized into four basic classes: Vector layers, graph layers, raster layers, and objects (agents and entities). Vector layers contain spatial information such as the position and structure of buildings, streets, or squares. Graph layers represent networks, for example, to model roads or public transport routes such as metro lines. Raster layers divide the considered space into equally large cells and store one or multiple – usually numerical – values per cell (e.g., the amount of rainfall). Agents are the active components of a simulation. Based on their environmental data, they are capable of autonomous actions and interact with each other to coordinate them. Such multi-agent interactions take place either directly via messages or indirectly via an environmental layer. Entities are not active, i.e., they cannot make decisions and initiate actions stand-alone, but have a life-cycle and can be used by agents (e.g., a car driven by a person).

After the initial state of the simulation has been created, the component starts the time-discrete simulation process. During this process, the simulation state is subject to a continuous change in each expiration of a previously defined step size (a so-called *tick*) with an optional real-time reference, e.g., layer values can change or agents can move. At the end of each tick, the current simulation state (including objects, layers, and tick metadata) is collected and passed to the output adapter, which forwards these results to their respective output channels (see below). State changes of individual agents, entities, and layers are synchronized to enable a consistent world state, i.e., all of them are always in the same tick. If the system

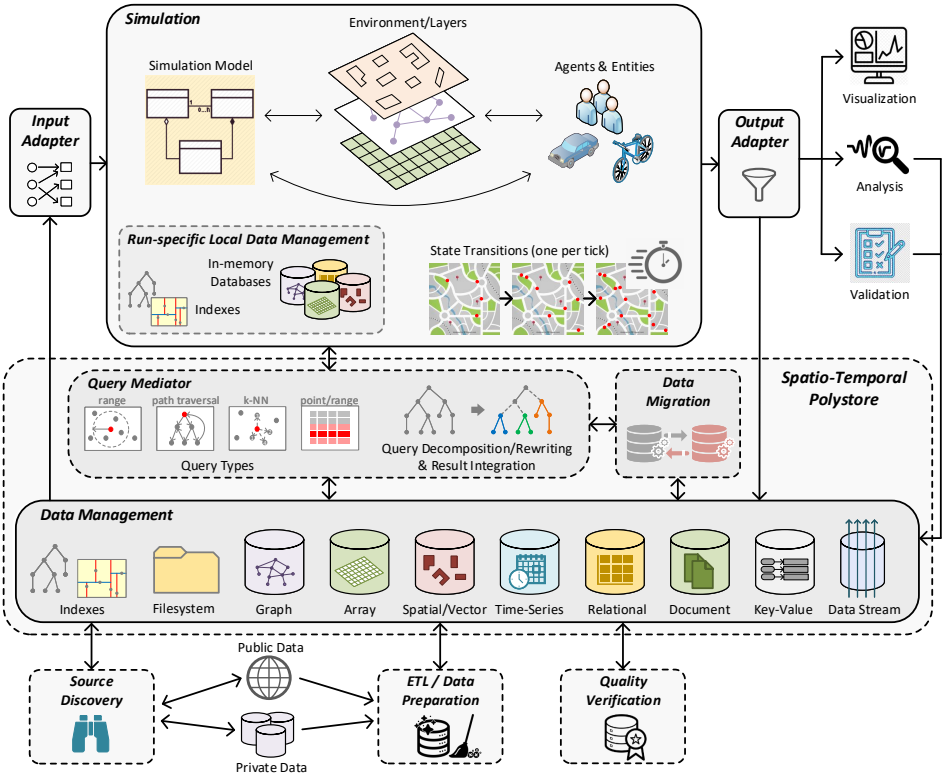


Fig. 1: Architecture of a Multi-Agent Simulation System

cannot load new data during runtime, the calculation of the following state can always only be based on the current one.

Data Management: The data management component includes various database systems of different data models, all of which serve a specific purpose. The user-defined agent-based models and some additional input files are persisted in a simple file system. Spatial rasters, e.g., for location-specific weather information, are suitable for data stores with array support (e.g., SciDB, Postgres, or Oracle GeoRaster). Vector-based features³, including houses, factories, and other points of interest, are stored in databases supporting spatial indexing (e.g., PostGIS or MongoDB). The temporal data management (e.g., Timescale or InfluxDB) concerns the validity periods and transactions of data objects as well as aspatial time-related data, e.g., business hours. Single or combinations of aggregate-oriented NoSQL or relational systems with graph abstraction or mapping collection, as applied by multi-model databases (e.g., ArangoDB or OrientDB), are suitable for storing domain data. This domain includes

³ A vector-based feature is a spatial geometry associated with a set of attributes and values.

entities and agents, each comprising subsets of value- or (un)directed reference-typed attributes (1:1, 1:n, or n:m) along inheritance hierarchies. Other facts or validity checks and analysis results can be stored in it as well. Finally, the input to the simulation system can be a data stream containing real-time data originating from sensor systems, such as temperature or air quality measurements. Additionally, the data management component manages several indexes, such as kd-trees for vector layers, that allows the input adapter and the query mediator (see below) fast access to the data.

The data management component has three subcomponents: The source discovery component is responsible for automatically detecting new and relevant datasets. Such a search can occur within a specific intranet (e.g., cloud), but also in the World Wide Web. The ETL/data preparation component loads external data (from private systems or the Web) into the data management component. Before loading it into an appropriate database, it standardizes, cleans, and enriches the data. The quality verification component is responsible for the high quality of the data already in the system. It includes a cross-database verification to detect inconsistencies between separately stored datasets.

Input Adapter: The input adapter supplies the runtime system with the simulation model (usually loaded from the file system) and initializes the first state as configured by the investigated scenario. This data mapping needs to overcome the impedance mismatch of the input data to the different kinds of models supported by the simulation component (i.e., vector, raster and graph layers as well as agent and entity classes). Therefore, it loads the data into local (in-memory) databases and indexes, kept for the duration of the currently running simulation process and – if implemented – are frequently updated by posing queries to the mediator (see below). These local databases allow fast access on layers, agents and entities, but limit the support only to point queries and k-NN queries with range filters. Unlike the query mediator, the input adapter is only used to map and load data to build the initial state of the simulation by utilizing user-defined scripts.

Query Mediator: In contrast to the input adapter, the query mediator enables dynamic and flexible ad-hoc access to the data management component by abstracting the required operations via a logical single query interface. Based on the mapping between defined operations and the underlying stores, the mediator decomposes queries into several subqueries while utilizing available store-specific features to make the most of their unique advantages. The subqueries are rewritten into native queries of the addressed stores and passed to them. Finally, their results are merged and forwarded to the running simulation or the user applying the analytical task. A rich query interface (including spatio-temporal operations and result formats), knowledge of store-specific features as well as planning, decomposing and rewriting queries are essential aspects in ensuring data transparency and providing short runtimes. The query mediator is supposed to support various types of queries, including: (i) spatio-temporal queries (e.g., to get objects from specific intersecting areas of Hamburg in the last month), (ii) range queries (e.g., to identify persons, cars, or buildings that are within a specific range to the ground zero of a disaster), (iii) k-NN queries (e.g., to identify the nearest bus stations or restaurants of a pedestrian), (iv) path traversals (e.g., to traverse

along the stations of a metro line), and (v) point queries (e.g., to access data of a particular point of interest). For practical reasons, k-NN queries must be combinable with range queries (i.e., the k-NN search is limited to a predefined range). Together with the data management and migration components, the query mediator forms a spatio-temporal polystore.

Data Migration: The structure and requirements for individual data objects change from time to time and often depend on the simulation processes using them. To meet such changing conditions, it can be useful to replicate data in different databases with different data models or migrate them from one model to another. Ideally, the system itself recognizes the demand for such a migration and automatically starts the corresponding migration process. Under certain circumstances, the query mediator initiates such a migration if it detects that the requested data are not available in the required format. In such a case, the migration must be performed at runtime either eagerly or lazily, for the current query only, or permanently. Since every migration step changes the location of the data, existing mappings between the databases and the simulation model may need to be updated.

Output Adapter: The output adapter is responsible to collect and forward snapshots of the individual objects and layers to the data management component and/or other software artifacts that aim to process them. Since the amount of data can be overwhelmingly large, exporting all of them can delay the simulation. Thus, it may be necessary that the adapter reduces the output to the most relevant values. It must also select a suitable format and compression method to keep the volume of data transferred as small as possible. Examples for relevant output data can be the volatile parts of the individual agents (e.g., position or vitality), but also the states of the different layers (e.g., temperature or water level) and additional measurements (e.g., traffic load).

Result Processing: Data exported by the output adapter can be stored directly in some of the databases, but can also be analyzed, visualized, and validated for violated constraints and expected behavior. The (often aggregated) results can, in turn, also be persisted in the databases. All three processing methods can be executed in real-time or batch mode, but only the first mode allows an intermediate intervention into the simulation's current state.

3 Challenges for Data Management

Besides the challenges that still need to be solved for polyglot data management in general [Pa16, Kr19, Ta17], such as query mediation [Cl98], automatic data migration [Kl16, SLD16] or cross-model replication [VSS18], there are a number of challenges that are specific to MAS systems. We will take a closer look at these challenges in this section.

Simulation Input: Although most simulations are limited to a specific area, it is almost always beneficial to transfer them to other areas by exchanging their location-specific data. For example when transferring a traffic simulation from Hamburg to Beijing, we need to exchange site maps, road networks and aspatial data, such as bus schedules. Since such a

location-based transfer is to be performed very often, flexibly and at short notice, adequate support in terms of (i) an *automatic discovery* and *acquisition* of relevant and qualitatively suitable input data, (ii) *preparation* of the newly acquired data, and (iii) an *automatic integration* of these data into the simulation model, is more relevant and crucial than in many other data integration use cases.

Dataset Discovery: Many spatio-temporal datasets are published in a structured form using a data portal/repository software, such as CKAN⁴ [As20] or dataverse [Te20]. To find useful data in those portals, we must first discover a suitable data portal and then search for the required data in it. To support the second step, CKAN and dataverse provide several functionalities including full-text search and fuzzy matching on the datasets' meta data as well as browsing between related datasets. In contrast, finding a suitable data portal is – to the best of our knowledge – currently not supported by any software. If we do not find the required data in any portal, we have no choice but to crawl the World Wide Web and to extract the – often unstructured – data from the found websites [Fu13, Fe14, Fa18].

Preparation: After loading the data from the discovered sources, we restrict them to the spatio-temporal range of the simulation by removing all data points that are outside the area and time period of the corresponding scenario. Thereafter, we need to standardize, clean and enrich the remaining data. Spatial standardization includes transformations into the same spatial reference system, such as UTM or USNG. Raster layers have to be transformed to the same scale and converted so that their cells are congruent. Graph layers need to be transformed into compatible graph models [AG08]. Timestamps have to be normalized by transforming them into the same format, calendar and time zone. Finally, the attribute data can be standardized using conventional preparation techniques [Py99, KJN20]. The data cleaning has to include a removal (or repair) of (i) spatial [CS06, KL17] and temporal [Gu14, Zh17] outliers, (ii) inconsistencies between different polygons (e.g., overlapping borders) or timestamps (e.g., a building was demolished before it was built), and (iii) errors in the attribute data, such as typos or violated dependencies [GS13, IC19, Ch14] where some of these errors can only be detected by comparing data from different layers. Useful examples of data enrichment include using alternative data sources to refine the road network of Open Street Map (OSM) [RFS16] or applying geocoding to locate address data in the spatial layers accurately [CCW04].

Integration: After collecting and preparing relevant source data, we need to integrate them into the simulation model. This includes resolving conflicts in the sources' spatial and temporal overlaps, such as contradicting geographical details (e.g., the same building is represented by different polygons). The integration can be materialized or virtual [LN07, DHI12]. In the first case, we initially integrate all source layers of the same type (raster, vector or graph) into a single layer persisted in the simulation system. Thus, all data conflicts are resolved in a typical ETL process [KR02] once before the simulation process starts. This approach, however, is static and cannot deal with real-world changes that happen when

⁴ Comprehensive Kerbal Archive Network

the simulation process is already running (see stream based input described below). In the second case, we integrate the source layers (and thus resolve conflicts) at query time by defining a global-as-view mapping [DHI12]. This is computationally more expensive, but flexible to changes against the source data. The same integration principles apply to temporal dimensions when we have several data sources covering different periods of time of the same spatial areas. To the best of our knowledge, there is currently no research that addresses a virtual integration of spatio-temporal data layers [GRC20].

Simulation Output: Exporting the snapshot of the current tick quickly becomes a bottleneck if the simulation is using a large number of agents. The biggest challenge is therefore to export the data without blocking the simulation process or creating a significant delay that would make it impossible to analyze, validate or visualize them in real time. The volume of the exported data is significantly related to whether we export only data changes or entire snapshots. The former reduces the volume, but makes immediate (possibly real-time) analyses, validations and visualizations of the simulation much more difficult because we need to reconstruct the actual snapshots from the exported changes.

Stream Based Input: The computed simulation states, including agent attributes and environment information, become fuzzier in their correctness as the simulation progresses and reaches further into the future. Public sensor data systems and APIs, such as the widely used SensorThingsAPI standard [LHK16], offer updates for temporally available environmental and entity-level information that can be used to reduce the corridor of uncertainty. Frequently updating the simulation states according to sensor-based input data by synchronizing the simulation with the real-time, results in a digital representation of real-world scenarios suitable for short-term forecasts and simplified global views of real-life happenings (e.g., to identify superspreading events within a pandemic). Problems are scalable handling of massive push-based inputs [WRG19] and merging incoming values on different time and granularity levels [CV86] without producing unrealistic simulation behavior (e.g., a full car park is emptied by beaming cars to remote locations). The latter can be done by either introducing them into the current simulation or forking a new one with the corrected state. Particularly relevant is the identification of model-independent growing uncertainties under consideration of user-defined constraints, defined via windowing queries and evaluated at the simulation's runtime. In addition to the usual integration problems, stream-based data present specific problems in dealing with non-equidistant inputs, duplicates, erroneous or noisy values, and sharp peaks. Solutions include the application of Kalman filters with wavelet corrections, comparisons of running windows for duplicate detection [SZ08, DNB13], and sliding aggregate functions [CV86]. For example, continuously averaging the attribute values of particular vector-based features can correct the simulation step by step.

Spatio-Temporal Query Interface: Because of the time-based definition of simulations, temporal operators are an essential aspect in the mediator's query interface. As it has been discussed by Siabato et al. [Si18], the support for Allen's interval operator [Al83] is essential for interval-based logical reasoning on time-series, getting versioned model objects. In context to the spatial characteristic, agents need access to environmental information for their

own decision making. This often corresponds to their current location, which requires k-NN queries with range filters. Such queries often have a circular shape, but can be abstracted to any geometrical shape. Polygon-based intersections require a pre-triangulation task to check for containment of the polygons' coordinates. Data access has to be provided by operators, such as *include*, *overlap* or *adjacent*, which also need to be part of the query interface [GRC20]. Since not only users perform analytical queries, but also the active agents themselves, the simulation can control queries against the mediator in time.

Spatio-Temporal Query Planning: The polystore has to manage the mapping between the simulation model's instance and its cross-system representation in the databases. This mapping requires a cross-system perspective, including requirements from the applied operations of the active agents in the simulation and subsequent analysis of results by the user. Populating the model with data from the polystore should be *independent* of the underlying databases and therefore transparent in the selection of convenient stores. Polyglot data storage offers the potential to meet a large set of (non-)functional requirements by taking into account the respective capabilities of each connected store in the mediated data processing. In order to exploit this potential for simulations, it is necessary to know the respective spatial, aspatial and temporal features of the individual databases and to describe them in a structured way. This description has to contain an input specification including constraints on expected objects as well as potentially produced outputs and their limitations. Query planning utilizes these feature descriptions to compute plans for distinct spatial and temporal queries, in which constraints are primarily affected by the expected models, for example, for relational data processing [SLD16]. Therefore, plans must address minimal intermediate migration steps where the cost of transfer does not exceed the cost of data processing. However, finding an optimal migration plan is NP-hard and can only be approximated [Kr19]. Beside migration problems, the system has to resolve references between data objects by providing an integration of partial results, either by implementing the bind-join [Ko16] approach or applying spatial-joins according to their references.

Further challenges concern cross-model data matching/merging [DHI12] and data lineage [HDB17] (e.g., to debug the simulation in case of errors).

4 Current Implementation

To meet the challenges of Section 3, we are extending our existing MARS architecture [GI17, WGC18, We19], which aims to provide large scale, agent-based simulations for any domain expert. The key idea behind MARS is to combine the flexibility of self-adaptive and data model agnostic simulation systems, by following an as-a-service perspective on modeling and simulation (MSaaS). MARS schedules and runs simulation (or optional other agent-systems) pods within a heterogeneous cluster environment and scale-up and out along available processing units and computing nodes.

Simulation Input: For the integration of new input data, MARS provides an external Python subsystem called ODDI ⁵ [G119]. ODDI integrates a CKAN, Open Data Protocol, and Open Street Map client. The system utilizes a keyword search on the portals and retrieves all metadata by using the MinHash similarity. Results include vector, raster and table meta data, loaded in memory through OpenGIS Web services (WFS, WCS for vector and WMS for raster data)⁶. ODDI can also be used to audit the datas' quality through statistic analyses using a small statistics package and integrated plotter. New spatial data is prepared by transforming it into the WGS:84 EPSG:4326 reference system. Timestamps are uniformly converted to the same format. To integrate spatio-temporal data, MARS uses a hybrid approach. While spatial data layers are integrated materially, temporal changes are integrated virtually (i.e., we manage a separate layer for every time period).

Simulation Output: Since the focus of MARS are large-scale scenarios, the calculated results are proportional to the dimensioned agent types with their respective number of instances per simulated tick [WGC18]. The system persists snapshots of agents and layers along the underlying databases according to the current workload. Collected snapshots are persisted either as complete object versions or as deltas from the last versions. Each persistence task is applied in a specified output frequency or if a model object has been changed since the last tick. In the data management component, it can be decided whether the results are fully replicated in all data stores, a subset of data stores is used in order to produce specific output formats, or all data are saved only in one store or file format. Due to wrong or missing semantics in the resulting data, not all output combinations are possible for every layer or agent type (e.g., in the case of a missing support for matrix types or raster files). In addition, the output can be reduce to specific states by using predefined *output conditions* (e.g., the current spatial extension coming from the visualization on the client map). In order to enable a fast and parallel transfer of the output data to analysis, validation and visualization tools, the data are passed to a Kafka pipeline.

Modelling & Querying: MARS uses a polyglot approach to data modeling. The entire platform supports the complete workflow of simulative analyses and offers the external static-typed MARS DSL modeling and query language [G117]. The language includes a type inference system and links the agent-based paradigm with the spatio-temporal layer. The language conceptualize type definitions (agent, entity, vector- and raster-layer) and allows spatial queries by applying *conditional area* filters and *k-NN* queries as well as access on *time series* by specifying concrete points in time.

When comparing our current solution to the challenges described in Section 3, the following differences become apparent: (i) The system accesses the data management component only via the in- and output adapter. (ii) No requests are delegated to the database by a mediator. Data is kept entirely in-memory during the simulation. (iii) Spatio-temporal queries are limited to in-memory indexes. (iv) Query planning is considered at compile-time and does not include online data migration. (v) ODDI allows for automated retrieval of public data and

⁵ Open Data Discovery and Integration

⁶ WFS = Web Feature Service, WCS = Web Coverage Service, WMS = Web Map Service

spatial linking, but the datas' quality has to be checked manually. (vi) Streaming data into a running simulation is currently not supported. We plan to fill these gaps by extending MARS to a spatio-temporal polystore [Ta17]. In developing the mediator, we plan to use the MARS DSL as the logical query interface. We intend to implement data migration by adapting current approaches [Kr19, HKS19] to our needs, including an extension to spatio-temporal features. Query planning is firstly accomplished by pushing-down operations to store-related features, applying selection queries for spatial or temporal data and integrating results via bind-joins. We will leverage existing research on web data extraction [Fa18] and data cleaning [Ch14] to improve ODDI. To realize an integration of data streams into a running simulation, we plan to evaluate several strategies for adapting simulation states to these real values, without producing significant anomalies in the simulation behaviour.

5 Related Work & Conclusion

The main goal of MARS is to support large-scale scenarios for general purposes by utilizing polyglot data management with spatial and temporal data processing. Other existing simulation systems, such as NetLogo [WR15] and GAMA [Gr13], focus on smaller-scaled scenarios with less complexity or involved agents. Although GAMA offers direct SQL database access to its agents, it does not consider a polyglot design and keeps transparency on the level of the SQL language. Yang et al. [Ya18] also use the layer concept for simulations, but do not consider temporal changes of spatial objects. The system of Zehe et al. [Ze16] involves multi-store data management and attempts to use each store appropriately for the tasks at hand, but lacks in making these decisions transparent and generic by integrating an automatic query planning component. In addition, the system does not allow spatio-temporal queries. Existing multi-/polystore systems, such as RHEEM [A119], Myria [Wa17], Polybase [De13] and ESTOCADA [A119], follow a general-purpose approach by unifying the query-interface or applying intermediate (self-defined or automatic) migration steps between stores, providing uniform read-only access. In our opinion, this approach is unsuitable for simulations, because it ignores change operations, processing queries with store-specific features, which is a major challenge in polyglot data management [Pa16, Ta17], and capabilities for querying spatio-temporal data. Systems, such as CloudMdSql [Ko16] and BigDAWG [Du15], are first promising candidates. CloudMdSql provides users with direct access to native data storage languages by embedding them into a SQL-like language, but lacks in providing data independence, so that the user must always know which data is stored in which data store. BigDAWG provides transparency at the level of multiple query languages, but does not support any kind of updates.

In this paper we presented the general architecture for data management in spatio-temporal multi-agent simulations. We concluded a number of challenges and gave a brief overview of the current status with open issues of our own system MARS. Future work will address the development of a query mediator as well as the (further) development of components for data migration, quality management, and real-time processing.

References

- [AG08] Angles, Renzo; Gutiérrez, Claudio: Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.
- [Al83] Allen, James F: Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.
- [Al19] Alotaibi, Rana; Bursztyn, Damian; Deutsch, Alin; Manolescu, Ioana; Zampetakis, Stamatis: Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, pp. 1660 – 1677, 2019.
- [As20] Association, CKAN: , CKAN – The Open Source Data Portal Software. <https://ckan.org/>, 2020. [Online; accessed 12-12-2020].
- [CCW04] Christen, Peter; Churches, Tim; Willmore, Alan: A Probabilistic Geocoding System based on a National Address File. In: *Proceedings of the 3rd Australasian Data Mining Conference*. 2004.
- [Ch14] Chiang, Yao-Yi; Wu, Bo; Anand, Akshay; Akade, Ketan; Knoblock, Craig A.: A System for Efficient Cleaning and Transformation of Geospatial Data Attributes. In: *Proceedings of the International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*. ACM, pp. 577–580, 2014.
- [Cl98] Cluet, Sophie; Delobel, Claude; Siméon, Jérundefinedme; Smaga, Katarzyna: Your Mediators Need Data Conversion! In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, p. 177–188, 1998.
- [CS06] Chawla, Sanjay; Sun, Pei: SLOM: A New Measure for Local Spatial Outliers. *Knowl. Inf. Syst.*, 9(4):412–429, 2006.
- [CV86] Chair, Z; Varshney, PK: Optimal Data Fusion in Multiple Sensor Detection Systems. *IEEE Transactions on Aerospace and Electronic Systems*, (1):98–101, 1986.
- [De13] DeWitt, David J.; Halverson, Alan; Nehme, Rimma; Shankar, Srinath; Aguilar-Saborit, Josep; Avanes, Artin; Flasz, Miro; Gramling, Jim: Split Query Processing in Polybase. In: *SIGMOD. SIGMOD '13*, Association for Computing Machinery, New York, New York, USA, p. 1255–1266, 2013.
- [DHI12] Doan, AnHai; Halevy, Alon; Ives, Zachary G.: *Principles of Data Integration*. Morgan Kaufmann, 2012.
- [DNB13] Dutta, Sourav; Narang, Ankur; Bera, Suman K.: Streaming Quotient Filter: A Near Optimal Approximate Duplicate Detection Approach for Data Streams. *Proc. VLDB Endow.*, 6(8):589–600, 2013.
- [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.
- [Fa18] Fayzrakhmanov, Ruslan R.; Sallinger, Emanuel; Spencer, Ben; Furche, Tim; Gottlob, Georg: Browserless Web Data Extraction: Challenges and Opportunities. In: *Proceedings of the International Conference on World Wide Web*. ACM, pp. 1095–1104, 2018.

- [Fe14] Ferrara, Emilio; Meo, Pasquale De; Fiumara, Giacomo; Baumgartner, Robert: Web Data Extraction, Applications and Techniques: A Survey. *Knowl. Based Syst.*, 70:301–323, 2014.
- [Fu13] Furche, Tim; Gottlob, Georg; Grasso, Giovanni; Schallhart, Christian; Sellers, Andrew Jon: OXPath: A Language for Scalable Data Extraction, Automation, and Crawling on the Deep Web. *VLDB J.*, 22(1):47–72, 2013.
- [Gl17] Glake, Daniel; Weyl, Julius; Dohmen, Carolin; Hüning, Christian; Clemen, Thomas: Modeling through Model Transformation with MARS 2.0. In: *ADS@SpringSim*. pp. 1–12, 2017.
- [Gl19] Glake, Daniel; Weyl, Julius; Lenfers, Ulfia A.; Clemen, Thomas: SmartOpenHamburg Verkehrssimulation: Automatisierte OpenData Integration für Multi-Agenten Simulation mit MARS. In: *Simulation in Umwelt- und Geowissenschaften*. 2019.
- [Gr13] Grignard, Arnaud; Taillandier, Patrick; Gaudou, Benoit; Vo, Duc An; Huynh, Nghi Quang; Drogoul, Alexis: GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In: *PRIMA*. pp. 117–131, 2013.
- [GRC20] Glake, Daniel; Ritter, Norbert; Clemen, Thomas: Utilizing Spatio-Temporal Data In Multi-Agent Simulation. unpublished, 2020.
- [GS13] Ganti, Venkatesh; Sarma, Anish Das: Data Cleaning: A Practical Perspective. *Synthesis Lectures on Data Management*. Morgan & Claypool Publishers, 2013.
- [Gu14] Gupta, Manish; Gao, Jing; Aggarwal, Charu C.; Han, Jiawei: Outlier Detection for Temporal Data: A Survey. *IEEE Trans. Knowl. Data Eng.*, 26(9):2250–2267, 2014.
- [HDB17] Herschel, Melanie; Diestelkämper, Ralf; Ben Lahmar, Houssem: A Survey on Provenance: What for? What form? What from? *VLDB J.*, 26(6):881–906, 2017.
- [HKS19] Holubová, Irena; Klettke, Meike; Störl, Uta: Evolution Management of Multi-model Data - (Position Paper). In: *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB Workshops, Poly and DMAH*. Springer, pp. 139–153, 2019.
- [IC19] Ilyas, Ihab F.; Chu, Xu: Data Cleaning. *ACM*, 2019.
- [KJN20] Koumarelas, Ioannis K.; Jiang, Lan; Naumann, Felix: Data Preparation for Duplicate Detection. *ACM J. Data Inf. Qual.*, 12(3):15:1–15:24, 2020.
- [Kl16] Klettke, Meike; Störl, Uta; Shenavai, Manuel; Scherzinger, Stefanie: NoSQL Schema Evolution and Big Data Migration at Scale. In: *IEEE Big Data*. pp. 2764–2774, 2016.
- [KL17] Kou, Yufeng; Lu, Chang-Tien: Outlier Detection, Spatial. In: *Encyclopedia of GIS*, pp. 1539–1546. Springer, 2017.
- [Ko16] Kolev, Boyan; Bondiombouy, Carlyna; Valduriez, Patrick; Jimenez-Peris, Ricardo; Pau, Raquel; Pereira, José: The CloudMdsQL Multistore System. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, p. 2113–2116, 2016.
- [KR02] Kimball, Ralph; Ross, Margy: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, 2nd Edition. Wiley, 2002.

- [Kr19] Kruse, Sebastian; Kaoudi, Zoi; Quiané-Ruiz, Jorge-Arnulfo; Chawla, Sanjay; Naumann, Felix; Contreras-Rojas, Bertty: Optimizing Cross-Platform Data Movement. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, pp. 1642–1645, 2019.
- [LHK16] Liang, Steve; Huang, Chih-Yuan; Khalafbeigi, Tania: OGC SensorThings API Part 1: Sensing, Version 1.0. 2016.
- [LN07] Leser, Ulf; Naumann, Felix: *Informationsintegration - Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. dpunkt.verlag, 2007.
- [LWC18] Lenfers, Ulfa A; Weyl, Julius; Clemen, Thomas: Firewood Collection in South Africa: Adaptive Behavior in Social-Ecological Models. *Land*, 7(3):97, 2018.
- [Pa16] Papakonstantinou, Yannis: Polystore Query Rewriting: The Challenges of Variety. In: *EDBT/ICDT Workshops*. 2016.
- [Py99] Pyle, Dorian: *Data Preparation for Data Mining*. Morgan Kaufmann, 1999.
- [RFS16] Richter, Andreas; Friedl, Hartmut; Scholz, Michael: Beyond OSM – Alternative Data Sources and Approaches Enhancing Generation of Road Networks for Traffic and Driving Simulations. In: *SUMO - Traffic, Mobility, and Logistics*. Deutsche Zentrum für Luft- und Raumfahrt, pp. 23–31, 2016.
- [Si18] Siabato, Willington; Claramunt, Christophe; Ilarri, Sergio; Manso-Callejo, Miguel Ángel: A Survey of Modelling Trends in Temporal GIS. *ACM Computing Surveys*, 51(2):1–41, 2018.
- [SLD16] Schildgen, Johannes; Lottermann, Thomas; DeBloch, Stefan: Cross-System NoSQL Data Transformations with NotaQL. In: *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR@SIGMOD)*. ACM, p. 5, 2016.
- [SZ08] Shen, Hong; Zhang, Yu: Improved Approximate Detection of Duplicates for Data Streams over Sliding Windows. *Journal of Computer Science and Technology*, 23(6):973–987, 2008.
- [Ta17] Tan, Ran; Chirkova, Rada; Gadepally, Vijay; Mattson, Timothy G: Enabling Query Processing across Heterogeneous Data Models: A Survey. In: *IEEE Big Data*. pp. 3211–3220, 2017.
- [Te20] Team, Dataverse: , The Dataverse Project – Open Source Research Data Repository Software. <https://dataverse.org/>, 2020. [Online; accessed 12-12-2020].
- [VSS18] Vogt, Marco; Stiemer, Alexander; Schuldt, Heiko: Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, New York, New York, USA, pp. 3364–3373, 2018.
- [Wa17] Wang, Jingjing; Baker, Tobin; Balazinska, Magdalena; Halperin, Daniel; Haynes, Brandon; Howe, Bill; Hutchison, Dylan; Jain, Shrainik; Maas, Ryan; Mehta, Parmita; Moritz, Dominik; Myers, Brandon; Ortiz, Jennifer; Suciu, Dan; Whitaker, Andrew; Xu, Shengliang: The Myria Big Data Management and Analytics System and Cloud Services. In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 2017.

- [Wa18] Waldrop, Mitchell: Free Agents - Monumentally Complex Models are Gaming out Disaster Scenarios with Millions of Simulated People. *Science*, 360(6385):144–147, 2018.
- [We19] Weyl, Julius; Lenfers, Ulfia A; Clemen, Thomas; Glake, Daniel; Panse, Fabian; Ritter, Norbert: Large-Scale Traffic Simulation for Smart City Planning with MARS. In: *SummerSim*. pp. 1–12, 2019.
- [WGC18] Weyl, Julius; Glake, Daniel; Clemen, Thomas: Agent-Based Traffic Simulation at City Scale with MARS. In: *ADS@SpringSim*. pp. 1–9, 2018.
- [WR15] Wilensky, Uri; Rand, William: *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. MIT Press, 2015.
- [WRG19] Wingerath, Wolfram; Ritter, Norbert; Gessert, Felix: *Real-Time & Stream Data Management - Push-Based Data in Research & Practice*. Springer Briefs in Computer Science. Springer, 2019.
- [Ya18] Yang, Liang Emlyn; Hoffmann, Peter; Scheffran, Jürgen; Rühle, Sven; Fischereit, Jana; Gasser, Ingenuin: An Agent-Based Modeling Framework for Simulating Human Exposure to Environmental Stresses in Urban Areas. *Urban Science*, 2(2):36, 2018.
- [Ye06] Yergens, Dean; Hiner, Julie; Denzinger, Jörg; Noseworthy, Tom: Multiagent Simulation System for Rapidly Developing Infectious Disease Models in Developing Countries. In: *MAS*BIOMED*. pp. 104–116, 2006.
- [Ze16] Zehe, Daniel; Viswanathan, Vaisagh; Cai, Wentong; Knoll, Alois: Online Data Extraction for Large-Scale Agent-Based Simulations. In: *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. pp. 69–78, 2016.
- [Zh17] Zhang, Aoqian; Song, Shaoxu; Wang, Jianmin; Yu, Philip S.: Time Series Data Cleaning: From Anomaly Detection to Anomaly Repairing. *Proc. VLDB Endow.*, 10(10):1046–1057, 2017.
- [ZKC05] Zaiyi, GUO; Kwang, HAN Hann; Cing, TAY Joc: Sufficiency Verification of HIV-1 Pathogenesis Based on Multi-Agent Simulation. In: *GECCO*. pp. 305–312, 2005.

Liste der Autorinnen und Autoren

A

Abedjan, Ziawasch, 313
Auge, Tanja, 337

B

Beer, Anna, 175
Binnig, Carsten, 325
Böhm, Alexander, 79
Brendle, Michael, 79

C

Clemen, Thomas, 423

D

D. Lohmann, 397
Dann, Jonas, 101
Decker, Stefan, 371

E

Eichler, Rebecca, 351
Esmailoghli, Mahdi, 313

F

Fischer, Stefan, 237
Franke, Martin, 257
Fröning, Holger, 101

G

Giebler, Corinna, 351
Glake, Daniel, 423
Gleim, Lars, 371
Gomez, Kevin, 303
Gröger, Christoph, 351
Groppe, Sven, 237
Grossniklaus, Michael, 79

H

Habich, Dirk, 135
Hagedorn, Stefan, 195
Hartmann, Claudio, 135
Herschel, Melanie, 155
Heuer, Andreas, 337
Hoos, Eva, 351

K

Karnowski, Lukas, 123
Kemper, Alfons, 39, 123
Kläbe, Steffen, 195
Kumaigorodski, Alexander, 19

L

Langenecker, Sven, 325
Lässig, Nico, 155
Lehner, Wolfgang, 135
Leis, Viktor, 39
Lenfers, Ulfa, 423
Lerm, Stefan, 217
Lucas, Edson R., 397
Lutz, Clemens, 19

M

Markl, Volker, 19, 279
Mauerer, Wolfgang, 397
May, Norman, 79
Mitschang, Bernhard, 351
Moerkotte, Guido, 79

N

Neumann, Thomas, 39, 123

O

Obermeier, Sandra, 175

Oppold, Sarah, 155
Özmen, Aslihan, 313

P

Panse, Fabian, 423
Papenbrock, Thorsten, 59
Paz, Elena Beatriz Ouro, 279

R

Rahm, Erhard, 217, 257, 303
Ramsauer, Ralf, 397
Rehan, Muhammad Waqas, 237
Ritter, Daniel, 101
Ritter, Norbert, 423
Rohde, Florens, 257
Rost, Christopher, 303
Rostami, M. Ali, 303

S

Saeedi, Alieh, 217
Schalles, Christian, 325
Scherzinger, Stefanie, 397
Schmeißer, Josef, 39
Schmidl, Sebastian, 59

Schüle, Maximilian E., 39, 123
Schulze, Robert, 79
Schwarz, Holger, 351
Sehili, Ziad, 257
Seidl, Thomas, 175
Sturm, Christoph, 325

T

Täschner, Matthias, 303
Tirpitz, Liam, 371

V

Valiyev, Mahammad, 79

W

Wahl, Florian, 175
Warnke, Benjamin, 237
Weber, Nick, 79
Weise, Julian, 59
Woltmann, Lucas, 135

Z

Zacharatou, Eleni Tzirita, 279